**Acta Informatica manuscript No.**
(will be inserted by the editor)

# Extracting Unsatisfiable Cores for LTL via Temporal Resolution

**Viktor Schuppan**

Received: / Accepted:

**Abstract** Unsatisfiable cores (UCs) are a well established means for debugging in a declarative setting. Still, there are few tools that perform automated extraction of UCs for LTL. Existing tools compute a UC as an unsatisfiable subset of the set of top-level conjuncts of an LTL formula. Using resolution graphs to extract UCs is common in other domains such as SAT. In this article we construct and optimize resolution graphs for temporal resolution as implemented in the temporal resolution-based solver TRP++, and we use them to extract UCs for propositional LTL. The resulting UCs are more fine-grained than the UCs obtained from existing tools because UC extraction also simplifies top-level conjuncts instead of treating them as atomic entities. For example, given an unsatisfiable LTL formula of the form $\phi \equiv (\mathbf{G}\psi) \wedge \mathbf{F}\psi'$ existing tools return $\phi$ as a UC irrespective of the complexity of $\psi$ and $\psi'$, whereas the approach presented in this article continues to remove parts not required for unsatisfiability inside $\psi$ and $\psi'$. Our approach also identifies groups of occurrences of a proposition that do not interact in a proof of unsatisfiability. We implement our approach in TRP++. Our experimental evaluation demonstrates that our approach (i) extracts UCs that are often significantly smaller than the input formula with an acceptable overhead and (ii) produces more fine-grained UCs than competing tools while remaining at least competitive in terms of run time and memory usage. The source code of our tool is publicly available.

**Keywords** LTL · unsatisfiable cores · resolution graphs · vacuity · temporal resolution

## 1 Introduction

### 1.1 Motivation

Debugging is an activity that many hardware and software developers spend a fair amount of time on. When faced with some input that induces an undesired behavior it is typically suggested to minimize that failure-inducing input in order to simplify identification of the

A preliminary version of this paper appeared in [85].

E-mail: Viktor.Schuppan@gmx.de
URL: http://www.schuppan.de/viktor/

problem (e.g., [98]). Corresponding research has been performed, e.g., in linear programming (e.g., [20]), constraint satisfaction (e.g., [6]), compilers (e.g., [97]), SAT (e.g., [16]), description logics (e.g., [81]), declarative specifications (e.g., [89]), and LTL satisfiability (e.g., [83]) and realizability (e.g., [23]).

LTL [36] and its relatives (e.g., [35, 63, 74]) are important specification languages for reactive systems (e.g., [35, 63]) and for business processes (e.g., [74]). Experience in verification as well as in synthesis has lead to specifications themselves becoming objects of analysis. Beer et al. report [9] that in their experience "[...] during the first formal verification runs of a new hardware design, typically 20 % of formulas are found to be trivially valid, and that trivial validity always points to a real problem in either the design or its specification or environment.". In a work on LTL synthesis [15] Bloem et al. state that "[...] writing a complete formal specification [...] was not trivial." and "Although this approach removes the need for verification [...] the specification itself still needs to be validated.".

Typically, a specification is expected to be satisfiable. If it turns out to be unsatisfiable, finding a reason for unsatisfiability can help with the ensuing debugging. Frequently, such a reason for unsatisfiability is taken to be a part of the unsatisfiable specification that is by itself unsatisfiable (e.g., [83, 6, 20]); this is called an unsatisfiable core (UC) (e.g., [83, 42, 100, 50]).

Less simplistic ways to examine an LTL specification $\phi$ exist [75], and understanding their results also benefits from availability of UCs. First, one can ask whether a certain scenario $\phi'$, given as an LTL formula, is permitted by $\phi$, i.e., whether in a situation, in which the specification $\phi$ holds, the scenario $\phi'$ can occur. That is the case iff $\phi \wedge \phi'$ is satisfiable. Second, one can check whether $\phi$ ensures a certain LTL property $\phi''$. $\phi''$ holds in $\phi$ iff $\phi \wedge \neg\phi''$ is unsatisfiable. In the first case, if the scenario turns out not to be permitted by the specification, a UC can help to understand which parts of the specification and the scenario are responsible for that. In the second case a UC can show which parts of the specification imply the property. Moreover, if there are parts of the property that are not part of the UC, then those parts of the property could be strengthened without falsifying the property in the specification; i.e., the property is vacuously satisfied (e.g., [9, 65, 2, 44, 92, 40, 64]).

UCs are therefore an important part of design methods for embedded systems (e.g., [75]) as well as for business processes (e.g., [3]). Note that specifications of real world systems may be 100s of pages long (e.g., [19]). Hence, providing automated support for obtaining a UC in case such a specification turns out to be unsatisfiable is crucial.

## 1.2 Contributions

**Fine-Grained UCs for LTL** Despite the relevance of UCs for LTL as outlined above interest in them has been somewhat limited (e.g., [22, 83, 3, 84, 43, 46, 47]). In particular, publicly available tools that automatically extract UCs for propositional LTL are scarce. We are aware of two such tools: `PLTL-MUP`[1] [43] and `procmine`[2] [3].

The UCs produced by `PLTL-MUP` and `procmine` are somewhat coarse-grained in the following sense. Both tools take as input a set of LTL formulas $\phi$.[3] If that set of LTL formulas is unsatisfiable, then they produce a subset $\phi^{uc} \subseteq \phi$ that is still unsatisfiable. However, they treat the LTL formulas that are the elements of $\phi$ as atomic entities; i.e.,

[1] http://www.timsergeant.com/pltl-mup/

[2] http://users.cecs.anu.edu.au/~rpg/BusinessProcessModelling/procmine.zip

[3] A set of LTL formulas $\phi$ is interpreted as the conjunction of all formulas in $\phi$.

they do not analyze whether all subformulas of the LTL formulas that make up $\phi^{uc}$ are required for unsatisfiability.

In this article we propose an approach that takes an LTL formula as input and determines for each node in the syntax tree whether the subformula rooted at that node is necessary for unsatisfiability. Hence, if $\phi \equiv \{\mathbf{G}(p \wedge \psi), \mathbf{F}((\neg p) \wedge \psi')\}$, then `PLTL-MUP` and `procmine` will return $\phi$ as UC irrespective of the complexity of $\psi$ and $\psi'$. Our approach takes $(\mathbf{G}(p \wedge \psi)) \wedge \mathbf{F}((\neg p) \wedge \psi')$ as input and returns $(\mathbf{G}(p \wedge \text{TRUE})) \wedge \mathbf{F}((\neg p) \wedge \text{TRUE})$ as UC.

**UCs for LTL via Temporal Resolution** Extracting UCs is often possible using any solver for the logic under consideration by weakening subformulas one by one and using the solver to test whether the weakened formula is still unsatisfiable (e.g., [68]). Although that is simple to implement, repeated testing for preservation of unsatisfiability may impose a significant run time burden. A potential alternative are methods that extract UCs by analyzing a single run of a solver. It is interesting to investigate such methods because they might not only reduce the run time burden but could also reveal additional information on why a formula is unsatisfiable (see, e.g., Sec. 5.2 and [84]). Extracting UCs from resolution graphs is common in SAT (e.g., [99]). A resolution method (e.g., [5]) for LTL, temporal resolution (TR), was suggested by Fisher [37, 39] and implemented in `TRP++` [54, 53]. `TRP++` is available as source code[4].

TR lends itself as a basis for extracting UCs for LTL for two reasons. First, the TR-based solver `TRP++` proved to be competitive in a recent evaluation of solvers for LTL satisfiability, in particular on unsatisfiable instances (see pp. 51–55 of the full version of [87]). Second, a TR proof naturally induces a resolution graph, which provides a clean framework for extracting a UC. Among the other solvers evaluated in [87] we mention the BDD-based solver `NuSMV` [21] and the tableau-based solvers `LWB` [49] and `pltl`[5]. Although `NuSMV` also performed well on unsatisfiable instances in [87], the BDD layer makes extraction of a UC more involved than a TR proof. On the other hand, `LWB` and `pltl` provide access to a proof of unsatisfiability comparable to TR, yet tended to perform worse than `TRP++` on unsatisfiable instances in [87].

In this article we show how to obtain a UC from an execution of the TR algorithm as implemented in `TRP++`. At the heart of our method is the construction of a resolution graph for TR for propositional LTL; note that TR is significantly more complex than propositional resolution. We also show how to use the specifics of TR in `TRP++` to optimize the construction of the resolution graph.

**Interaction of Occurrences of Propositions in a UC** A resolution graph contains not only information on which parts of the input formula were used to derive unsatisfiability but also how these parts were used. We therefore suggest to exploit the resolution graph to provide more detailed information on unsatisfiability. In this article we use the resolution graph to point out which occurrences of atomic propositions interact in a UC. In a companion paper [84], which this article provides the basis for, we use it to show which subformulas are relevant for unsatisfiability at which points in time.

**Mapping a UC in Separated Normal Form to a UC in LTL** A potential disadvantage of using TR for extracting UCs for LTL is the fact that TR does not work directly on LTL but on a clausal normal form called Separated Normal Form (SNF) [37, 38, 39]. Translations from an LTL formula into an equisatisfiable formula in a clausal normal form are well known both in temporal resolution (e.g., [37, 38, 39]) and in (symbolic)

[4] http://www.csc.liv.ac.uk/~konev/software/trp++/

[5] http://users.cecs.anu.edu.au/~rpg/PLTLProvers/

model checking (e.g., [66, 18, 26, 60, 13]). However, for optimal support of a user who tries to track down the source of the unsatisfiability of a formula it is likely to be helpful if the UC that is presented to her is "syntactically close" to the formula that she provided as an input to the solver. Hence, it is necessary to map a UC obtained in SNF back to LTL. We show how to translate a UC from SNF back to the LTL formula that the user provided as an input.

**Reductions Between UCs for LTL and Mutual Vacuity** We discuss the relation between UCs for LTL and mutual vacuity [44]. Specifically, we prove that the problem of mutual vacuity for a system and a specification described by an LTL formula is reducible to the problem of finding a UC for an LTL formula and vice versa.

**Publicly Available Implementation** We implement our method in `TRP++`. We make the source code of our solver publicly available.

**Experimental Evaluation** Our experimental evaluation demonstrates that our approach (i) is viable in terms of the run time and memory overhead that it induces, (ii) can significantly reduce the size of an unsatisfiable input formula, and (iii) is competitive to alternative approaches, while it produces more fine-grained results.

UCs also have applications in avoiding the exploration of parts of a search space that can be known not to contain a solution for reasons "equivalent" to the reasons for previous failures (e.g., [28, 22]) and in certifying the correctness of a result of unsatisfiability (e.g., [96, 42, 100]). These applications also benefit from our results. Nevertheless, we only focus on debugging.

### 1.3 Related Work

As discussed above we are aware of two tools that compute UCs for LTL: `PLTL-MUP` and `procmine`; both produce UCs that are less fine-grained than the ones obtained with our approach. They also rely on different algorithms to determine the satisfiability of an LTL formula and to subsequently extract a UC from an unsatisfiable formula. `PLTL-MUP`, which appeared after an early version of this work [82] was made public, applies an approach to determine minimal UCs by Huang for SAT [52] to a BDD-based solver for LTL. `procmine` extracts UCs as part of a tool set for synthesizing business process templates. It uses a tableau-based solver to obtain an initial subset of an unsatisfiable set of LTL formulas and then applies deletion-based minimization to that subset. In principle also tools to determine mutual vacuity for LTL can be used to obtain UCs for LTL. However, none of the tools we considered turned out to be suitable for that task in practice. For details see Sec. 6.

In [22] Cimatti et al. perform extraction of UCs for PSL to accelerate a PSL satisfiability solver by performing Boolean abstraction. Their notion of UCs is coarser than ours and their solver is based on BDDs and on SAT. An investigation of notions of UCs for LTL including the relation between UCs and vacuity is performed by the author in [83]. Various notions based on syntax trees, on conjunctive normal forms, on tableaux, and on SAT-based bounded model checking (e.g., [12]) are discussed. One of the translations from LTL into a conjunctive normal form is very similar to the one used in this article. It is accompanied by a translation back from the conjunctive normal form to an LTL formula, but it assumes a minimal UC as its input. In some cases it provides more information than the translation from SNF back to LTL used in this article. For example, it points out that a positive polarity occurrence of an until formula can be replaced with a weak until formula. That feature is left as future work in this article. No implementation or experimental results are reported, and TR

is not considered. Hantry et al. suggest a method to extract UCs for LTL in a tableau-based solver [46]. No implementation or experiments are reported. In [47] the decision and search problems for minimal UCs for LTL are shown to be PSPACE- and FPSPACE-complete, respectively. In [25] Cimatti et al. show how to prove and explain unfeasibility of message sequence charts for networks of hybrid automata. They consider a different specification language and use an SMT-based (e.g., [7]) algorithm.

Some work deals with unrealizable rather than unsatisfiable cores. [23] handles specifications in GR(1), which is a proper subset of LTL. Könighofer et al. present methods to help debugging unrealizable specifications by extracting unrealizable cores and simulating counterstrategies [61] as well as performing error localization using model-based diagnosis [62]. Raman and Kress-Gazit [78] present a tool that points out unrealizable cores in the context of robot control. [83] explores more fine-grained notions of unrealizable cores than [23, 61].

In vacuity Simmonds et al. [92] use SAT-based bounded model checking for vacuity detection. They only consider $k$-step vacuity, i.e., taking into account bounded model checking runs up to a bound $k$, and leave the problem of removing the bound $k$ open. They use a similar scheme as described in Sec. 5.2 on proofs of unsatisfiability of propositional formulas to determine whether some proposition is vacuous. They only distinguish whether in a proof of unsatisfiability of a bounded model checking instance any occurrence of a given proposition in the specification interacts with any occurrence of that proposition in the system or not. Armoni et al. [2] discuss vacuity of an LTL specification in different polarity occurrences of subformulas. For a more extensive discussion on the relation between vacuity and UCs for LTL we refer to Sec. 6 and to [83].

Many algorithmic considerations are shared between UCs for LTL and UCs for other logics. Here we mention some work that can serve as starting point for studying UCs for other logics as well as work that contains ideas that might be applicable also to UCs for LTL. Extensive research has been performed on UCs for SAT. For a brief overview of early algorithms see, e.g., [70], pp. 68–70. Surveys of different aspects are, e.g., [68, 69, 17]. Early work on UCs for SMT was performed by Cimatti et al. in [24]. Pointers to work on modifying proofs in order to obtain smaller proofs and/or different interpolants (for some references see, e.g., [34]) in the propositional or SMT cases can be found in [79]. Some of this work traces which occurrences of literals were resolved with each other (e.g., [1]). Belov and Marques-Silva [11] and Shlyakhter [88] investigate UCs for circuits, i.e., formulas with potential sharing of subformulas rather than sets of clauses. UCs obtained from a SAT solver are also used to support debugging in the declarative modeling language Alloy [89, 95]. This requires translation from Alloy into an input for a SAT solver and back [89]; while the description in [89] is somewhat abstract, some basic ideas regarding the translation from Alloy into the input for the SAT solver and back are similar to ideas used in our translation. Also in their case a minimal UC obtained from the SAT solver does not guarantee a minimal UC in Alloy. [95] evaluates schemes to increase the efficiency of the UC extraction method. In description logics (e.g., [4]) UCs are frequently used to support knowledge engineers in performing their tasks. In particular, understanding, debugging, and possibly repairing entailments in an ontology can be aided by providing the user with subsets of axioms of an ontology that justify a given entailment. Some work also considers parts of axioms, leading to more fine-grained results. For an overview see, e.g., [51].

### 1.4 Structure of the Article

We start in Sec. 2 by providing examples that illustrate how UCs are useful for debugging. In Sec. 3 the more formal exposition begins with preliminaries. In Sec. 4 we describe the construction and optimization of a resolution graph and its use to obtain a UC. In Sec. 5 we adapt two methods to post-process the UCs obtained to make them more useful. The relation between UCs for LTL and mutual vacuity is discussed in Sec. 6. We present our implementation and experimental evaluation in Sec. 7. In Sec. 8 we draw conclusions.

Due to space constraints a few more involved parts of a proof as well as some more detailed data from our experimental evaluation are omitted. Both are included in the full version [86] of this article, which can be obtained from `http://www.schuppan.de/viktor/actainformatica15/` along with implementation, examples, and log files.

## 2 Motivating Examples

In this section we present examples of using UCs for LTL to help understand why a specification given in LTL is unsatisfiable. As we formally introduce LTL only in Sec. 3.1, those unfamiliar with LTL are asked to jump ahead to Sec. 3.1 first. We start with a toy example and then proceed to a more realistic one. Except for minor rewriting, all UCs in this section were obtained with our implementation.

### 2.1 Toy Example

The first example (1a)–(1c) is based on [56]. (1a) requires a *req* (request) to be followed by three *gnt*s (grant). In contrast, (1b) forbids two subsequent *gnt*s. (1c) states that from the time point after a *cancel* no *gnt* may be issued until a *go* is received. We would like to see whether a *req* can eventually be issued (1d).

$$(\mathbf{G}(req \rightarrow ((\mathbf{X}gnt) \wedge (\mathbf{XX}gnt) \wedge \mathbf{XXX}gnt))) \tag{1a}$$
$$\wedge\ (\mathbf{G}(gnt \rightarrow \mathbf{X}\neg gnt)) \tag{1b}$$
$$\wedge\ (\mathbf{G}(cancel \rightarrow \mathbf{X}((\neg gnt)\mathbf{U}go))) \tag{1c}$$
$$\wedge\ \mathbf{F}req \tag{1d}$$

Clearly, (1) is unsatisfiable. If a *req* were issued in (1d), (1a) would trigger three subsequent *gnt*s. However, already the second of those *gnt*s would be forbidden by (1b). Note that in this reasoning neither the third *gnt* in (1a) nor (1c) play a role. Hence, (1) would be unsatisfiable even if $\mathbf{XXX}gnt$ and $(\mathbf{G}(cancel \rightarrow \mathbf{X}((\neg gnt)\mathbf{U}go)))$ were "removed". The UC in (2) does just that by replacing these two subformulas with TRUE and simplifying. Note that in (2) not only a whole top-level conjunct has been removed from (1) but also a proper subformula inside a top-level conjunct.

$$(\mathbf{G}(req \rightarrow ((\mathbf{X}gnt) \wedge \mathbf{XX}gnt))) \wedge (\mathbf{G}(gnt \rightarrow \mathbf{X}\neg gnt)) \wedge \mathbf{F}req \tag{2}$$

### 2.2 Lift Specification

The second example (3) in Fig. 1 is adapted from a lift specification in [48]. The lift has two floors, indicated by $f_0$ and $f_1$. On each floor there is a button to call the lift ($b_0$, $b_1$). *sb*

$$
\begin{aligned}
&(\neg u) \wedge f_0 \wedge (\neg b_0) \wedge (\neg b_1) \wedge (\neg up) && \text{(3a)}\\
\wedge\ &(\mathbf{G}((u \rightarrow \neg \mathbf{X} u) \wedge ((\neg \mathbf{X} u) \rightarrow u))) && \text{(3b)}\\
\wedge\ &(\mathbf{G}(f_0 \rightarrow \neg f_1)) && \text{(3c)}\\
\wedge\ &(\mathbf{G}((f_0 \rightarrow \mathbf{X}(f_0 \vee f_1)) \wedge (f_1 \rightarrow \mathbf{X}(f_0 \vee f_1)))) && \text{(3d)}\\
\wedge\ &(\mathbf{G}(u \rightarrow ((f_0 \rightarrow \mathbf{X} f_0) \wedge ((\mathbf{X} f_0) \rightarrow f_0) \wedge (f_1 \rightarrow \mathbf{X} f_1) \wedge ((\mathbf{X} f_1) \rightarrow f_1)))) && \text{(3e)}\\
\wedge\ &(\mathbf{G}((\neg u) \rightarrow ((b_0 \rightarrow \mathbf{X} b_0) \wedge ((\mathbf{X} b_0) \rightarrow b_0) \wedge (b_1 \rightarrow \mathbf{X} b_1) \wedge ((\mathbf{X} b_1) \rightarrow b_1)))) && \text{(3f)}\\
\wedge\ &(\mathbf{G}(((b_0 \wedge \neg f_0) \rightarrow \mathbf{X} b_0) \wedge ((b_1 \wedge \neg f_1) \rightarrow \mathbf{X} b_1))) && \text{(3g)}\\
\wedge\ &(\mathbf{G}((f_0 \wedge \mathbf{X} f_0) \rightarrow ((up \rightarrow \mathbf{X} up) \wedge ((\mathbf{X} up) \rightarrow up)))) && \text{(3h)}\\
\wedge\ &(\mathbf{G}((f_1 \wedge \mathbf{X} f_1) \rightarrow ((up \rightarrow \mathbf{X} up) \wedge ((\mathbf{X} up) \rightarrow up)))) && \text{(3i)}\\
\wedge\ &(\mathbf{G}(((f_0 \wedge \mathbf{X} f_1) \rightarrow up) \wedge ((f_1 \wedge \mathbf{X} f_0) \rightarrow \neg up))) && \text{(3j)}\\
\wedge\ &(\mathbf{G}((sb \rightarrow (b_0 \vee b_1)) \wedge ((b_0 \vee b_1) \rightarrow sb))) && \text{(3k)}\\
\wedge\ &(\mathbf{G}((f_0 \wedge \neg sb) \rightarrow (f_0 \mathbf{U} (sb \mathbf{R} ((\mathbf{F} f_0) \wedge \neg up))))) && \text{(3l)}\\
\wedge\ &(\mathbf{G}((f_1 \wedge \neg sb) \rightarrow (f_1 \mathbf{U} (sb \mathbf{R} ((\mathbf{F} f_0) \wedge \neg up))))) && \text{(3m)}\\
\wedge\ &(\mathbf{G}((b_0 \rightarrow \mathbf{F} f_0) \wedge (b_1 \rightarrow \mathbf{F} f_1))) && \text{(3n)}
\end{aligned}
$$

**Fig. 1** A lift specification

is TRUE if some button is pressed. If the lift moves up, then $up$ must be TRUE; if it moves down, then $up$ must be FALSE. $u$ switches turns between actions by users of the lift ($u$ is TRUE) and actions by the lift ($u$ is FALSE). For more details we refer to [48].

We first assume that an engineer is interested in seeing whether it is possible that $b_1$ is pressed at time point 0 (4). As the UC (5) shows, this is impossible because $b_1$ must be FALSE at the beginning. (5) was obtained by conjoining (3) with (4), determining unsatisfiability, replacing all top-level conjuncts except for $\neg b_1$ and $b_1$ with TRUE, and simplifying.

$$b_1 \tag{4}$$

$$(\neg b_1) \wedge b_1 \tag{5}$$

Now the engineer modifies her query such that $b_1$ is pressed at time point 1 (6). As shown by the UC in (7) that turns out to be impossible, too. Note that while in the previous scenario (4) unsatisfiability was not too hard to see even without UC extraction, in the current scenario (6) UC extraction already is quite helpful in localizing which parts of (3) and (6) are responsible for unsatisfiability. The UC in (7) was obtained by conjoining (3) with (6), determining unsatisfiability, replacing all top-level conjuncts except for $\neg u$, $\neg b_1$, (3f), and $\mathbf{X} b_1$ with TRUE, replacing $b_0 \rightarrow \mathbf{X} b_0$, $(\mathbf{X} b_0) \rightarrow b_0$, $b_1 \rightarrow \mathbf{X} b_1$ in (3f) with TRUE, and simplifying. As in the toy example in the previous subsection not only top-level conjuncts but also proper subformulas inside a top-level conjunct have been simplified.

$$\mathbf{X} b_1 \tag{6}$$

$$(\neg u) \wedge (\neg b_1) \wedge (\mathbf{G}((\neg u) \rightarrow ((\mathbf{X} b_1) \rightarrow b_1))) \wedge \mathbf{X} b_1 \tag{7}$$

The engineer now tries to have $b_1$ pressed at time point 2 and, again, obtains a UC. She becomes suspicious and checks whether $b_1$ can be pressed at all (8). The unsatisfiability of the conjunction of (3) and (8) tell her that $b_1$ cannot be pressed at all and, therefore, this specification of a lift must contain a bug. She can now use the UC in (9) to track down

the problem. This example clearly illustrates the use of UCs for debugging as (9a)–(9f) is significantly smaller than (3).

$$\mathbf{F} b_1 \tag{8}$$

$$\begin{aligned}
& f_0 \wedge (\neg b_1) \wedge (\neg up) && \text{(9a)}\\
\wedge\ & (\mathbf{G}(f_0 \rightarrow \neg f_1)) && \text{(9b)}\\
\wedge\ & (\mathbf{G}(f_0 \rightarrow \mathbf{X}(f_0 \vee f_1))) && \text{(9c)}\\
\wedge\ & (\mathbf{G}((f_0 \wedge \mathbf{X} f_0) \rightarrow ((\mathbf{X} up) \rightarrow up))) && \text{(9d)}\\
\wedge\ & (\mathbf{G}((f_0 \wedge \mathbf{X} f_1) \rightarrow up)) && \text{(9e)}\\
\wedge\ & (\mathbf{G}(b_1 \rightarrow \mathbf{F} f_1)) && \text{(9f)}\\
\wedge\ & \mathbf{F} b_1 && \text{(9g)}
\end{aligned}$$

## 3 Preliminaries

In this section we first present LTL (Sec. 3.1). We continue with SNF, the clausal normal form for LTL that the temporal resolution algorithm works on, (Sec. 3.2) and with a translation from LTL into SNF (Sec. 3.3). Finally, in Sec. 3.4 we discuss the temporal resolution algorithm that our method for UC extraction in Sec. 4 is based on.

### 3.1 LTL

We use a standard version of LTL, see, e.g., [36]. Let $\mathbb{B}$ be the set of Booleans, and let $AP$ be a finite set of atomic propositions. The set of *LTL formulas* is constructed inductively as follows. The Boolean constants FALSE, TRUE $\in \mathbb{B}$ and any atomic proposition $p \in AP$ are LTL formulas. If $\psi$, $\psi'$ are LTL formulas, so are $\neg\psi$ (not), $\psi \vee \psi'$ (or), $\psi \wedge \psi'$ (and), $\mathbf{X}\psi$ (next time), $\psi \mathbf{U} \psi'$ (until), $\psi \mathbf{R} \psi'$ (releases), $\mathbf{F}\psi$ (finally), and $\mathbf{G}\psi$ (globally). We use $\psi \rightarrow \psi'$ (implies) as an abbreviation for $(\neg\psi) \vee \psi'$. An occurrence of a subformula $\psi$ of an LTL formula $\phi$ has *positive polarity* $(+)$ if it appears under an even number of negations in $\phi$ and *negative polarity* $(-)$ otherwise. The *size* of an LTL formula $\phi$ is measured as the sum of the numbers of occurrences of atomic propositions, Boolean operators, and temporal operators in $\phi$.

LTL is interpreted over words in $(2^{AP})^\omega$. For the semantics of LTL see Fig. 2. A word $\pi \in (2^{AP})^\omega$ *satisfies* an LTL formula $\phi$ iff $(\pi, 0) \models \phi$. A word $\pi$ that satisfies $\phi$ is also called a *satisfying assignment* for $\phi$. An LTL formula $\phi$ is *satisfiable* if there exists a word $\pi \in (2^{AP})^\omega$ that satisfies $\phi$; otherwise, it is *unsatisfiable*. The problem of determining the satisfiability of an LTL formula is PSPACE-complete [93, 45], even if the set of atomic propositions $AP$ only contains one element [30].

### 3.2 Separated Normal Form

Temporal resolution works on formulas in a clausal normal form called Separated Normal Form (SNF) [37, 38, 39]. For any atomic proposition $p \in AP$ $p$ and $\neg p$ are *literals*.

$$
\begin{array}{lcl}
(\pi,i) \models \text{TRUE} & & \\
(\pi,i) \not\models \text{FALSE} & & \\
(\pi,i) \models p & \Leftrightarrow & p \in \pi[i] \\
(\pi,i) \models \neg\psi & \Leftrightarrow & (\pi,i) \not\models \psi \\
(\pi,i) \models \psi \vee \psi' & \Leftrightarrow & (\pi,i) \models \psi \text{ or } (\pi,i) \models \psi' \\
(\pi,i) \models \psi \wedge \psi' & \Leftrightarrow & (\pi,i) \models \psi \text{ and } (\pi,i) \models \psi' \\
(\pi,i) \models \mathbf{X}\psi & \Leftrightarrow & (\pi,i+1) \models \psi \\
(\pi,i) \models \psi\mathbf{U}\psi' & \Leftrightarrow & \exists i' \geq i \,.\, ((\pi,i') \models \psi' \wedge \forall i \leq i'' < i' \,.\, (\pi,i'') \models \psi) \\
(\pi,i) \models \psi\mathbf{R}\psi' & \Leftrightarrow & \forall i' \geq i \,.\, ((\pi,i') \models \psi' \vee \exists i \leq i'' < i' \,.\, (\pi,i'') \models \psi) \\
(\pi,i) \models \mathbf{F}\psi & \Leftrightarrow & \exists i' \geq i \,.\, (\pi,i') \models \psi \\
(\pi,i) \models \mathbf{G}\psi & \Leftrightarrow & \forall i' \geq i \,.\, (\pi,i') \models \psi
\end{array}
$$

**Fig. 2** Semantics of LTL. $\pi$ is a word in $(2^{AP})^\omega$, $i$ is a time point in $\mathbb{N}$.

Let $p_1,\ldots,p_n$, $q_1,\ldots,q_{n'}$, $l$ with $0 \leq n,n'$ be literals such that $\forall 1 \leq i < i' \leq n \,.\, p_i \neq p_{i'}$ and $\forall 1 \leq i < i' \leq n' \,.\, q_i \neq q_{i'}$. Then (i) $(p_1 \vee \ldots \vee p_n)$ is an *initial clause*; (ii) $(\mathbf{G}(p_1 \vee \ldots \vee p_n \vee \mathbf{X}(q_1 \vee \ldots \vee q_{n'})))$ is a *global clause*; and (iii) $(\mathbf{G}(p_1 \vee \ldots \vee p_n \vee \mathbf{F}l))$ is an *eventuality clause*. $l$ is called an *eventuality literal*. As usual an empty disjunction (resp. conjunction) stands for FALSE (resp. TRUE). $()$ or $(\mathbf{G}())$, denoted $\square$, stand for FALSE or $\mathbf{G}(\text{FALSE})$ and are called *empty clause*. The set of all SNF clauses is denoted $\mathbb{C}$. Let $c_1,\ldots,c_n$ with $0 \leq n$ be SNF clauses. Then $\bigwedge_{1\leq i\leq n} c_i$ is an LTL formula in *SNF*. Every LTL formula $\phi$ can be transformed into an equisatisfiable formula $\phi'$ in SNF [39].

### 3.3 Translating LTL into SNF

We use a structure-preserving translation (e.g., [76]) to translate an LTL formula into a set of SNF clauses. Our translation is based on the tableau construction for LTL that is often used in (symbolic) model checking (see, e.g., [66, 18, 26, 60, 13]) rather than on [39] as we find the former to be more straightforward.

**Definition 1 (Translation from LTL into SNF)** Let $\phi$ be an LTL formula over atomic propositions $AP$, and let $X = \{x,x',\ldots\}$ be a set of fresh atomic propositions that don't occur in $\phi$. Assign to each occurrence of a subformula $\psi$ in $\phi$ a Boolean value or a proposition according to column 2 of Tab. 1, which is used to reference $\psi$ in the SNF clauses for its superformula. Moreover, assign to each occurrence of $\psi$ a set of SNF clauses according to columns 3 and 4 of Tab. 1. Let $SNF_{aux}(\phi)$ be the set of all SNF clauses obtained from $\phi$ that way. Then the *SNF of* $\phi$ is defined as $SNF(\phi) \equiv x_\phi \wedge \bigwedge_{c\in SNF_{aux}(\phi)} c$.

Note that to make the SNF clauses in column 4 of Tab. 1 and elsewhere in this article easier to understand we often use implication to formulate them. However, in TRP++ SNF clauses cannot contain implications and, therefore, in our implementation of Def. 1 implication is expanded using its definition. The fact that some propositions are marked blue boxed in Tab. 1 will be used later in Sec. 4.2 when translating a UC back from SNF to LTL. It is well known that $\phi$ and $SNF(\phi)$ are equisatisfiable and that a satisfying assignment for $\phi$ (resp. $SNF(\phi)$) can be extended (resp. restricted) to a satisfying assignment for $SNF(\phi)$ (resp. $\phi$). Below we sometimes identify the SNF of $\phi$, $SNF(\phi)$, with the set of SNF clauses $\{x_\phi\} \cup SNF_{aux}(\phi)$ that $SNF(\phi)$ is constructed from.

*Remark 1 (Complexity Considerations Regarding the Translation from LTL into SNF)* Let $\phi$ be an LTL formula over atomic propositions $AP$, and let $SNF(\phi)$ be the SNF of $\phi$. It is easy to see that (i) the number of clauses in $SNF(\phi)$ is linear in the size of $\phi$, (ii) for each

**Table 1** Translation from LTL into SNF

| Subformula | Proposition | Polarity | SNF Clauses |
|---|---|---|---|
| TRUE/FALSE/$p$ | TRUE/FALSE/$p$ | $+/-$ | none |
| $\neg\psi$ | $x_{\neg\psi}$ | $+$ | $(\mathbf{G}(x_{\neg\psi} \rightarrow \neg\boxed{x_\psi}))$ |
| | | $-$ | $(\mathbf{G}((\neg x_{\neg\psi}) \rightarrow \boxed{x_\psi}))$ |
| $\psi \vee \psi'$ | $x_{\psi\vee\psi'}$ | $+$ | $(\mathbf{G}(x_{\psi\vee\psi'} \rightarrow (\boxed{x_\psi} \vee \boxed{x_{\psi'}})))$ |
| | | $-$ | $(\mathbf{G}((\neg x_{\psi\vee\psi'}) \rightarrow \neg\boxed{x_\psi}))$, $(\mathbf{G}((\neg x_{\psi\vee\psi'}) \rightarrow \neg\boxed{x_{\psi'}}))$ |
| $\psi \wedge \psi'$ | $x_{\psi\wedge\psi'}$ | $+$ | $(\mathbf{G}(x_{\psi\wedge\psi'} \rightarrow \boxed{x_\psi}))$, $(\mathbf{G}(x_{\psi\wedge\psi'} \rightarrow \boxed{x_{\psi'}}))$ |
| | | $-$ | $(\mathbf{G}((\neg x_{\psi\wedge\psi'}) \rightarrow ((\neg\boxed{x_\psi}) \vee \neg\boxed{x_{\psi'}})))$ |
| $\mathbf{X}\psi$ | $x_{\mathbf{X}\psi}$ | $+$ | $(\mathbf{G}(x_{\mathbf{X}\psi} \rightarrow \mathbf{X}\boxed{x_\psi}))$ |
| | | $-$ | $(\mathbf{G}((\neg x_{\mathbf{X}\psi}) \rightarrow \mathbf{X}\neg\boxed{x_\psi}))$ |
| $\psi\mathbf{U}\psi'$ | $x_{\psi\mathbf{U}\psi'}$ | $+$ | $(\mathbf{G}(x_{\psi\mathbf{U}\psi'} \rightarrow (\boxed{x_{\psi'}} \vee \boxed{x_\psi})))$, $(\mathbf{G}(x_{\psi\mathbf{U}\psi'} \rightarrow (\boxed{x_{\psi'}} \vee \mathbf{X}x_{\psi\mathbf{U}\psi'})))$, $(\mathbf{G}(x_{\psi\mathbf{U}\psi'} \rightarrow \mathbf{F}\boxed{x_{\psi'}}))$ |
| | | $-$ | $(\mathbf{G}((\neg x_{\psi\mathbf{U}\psi'}) \rightarrow \neg\boxed{x_{\psi'}}))$, $(\mathbf{G}((\neg x_{\psi\mathbf{U}\psi'}) \rightarrow ((\neg\boxed{x_\psi}) \vee \mathbf{X}\neg x_{\psi\mathbf{U}\psi'})))$ |
| $\psi\mathbf{R}\psi'$ | $x_{\psi\mathbf{R}\psi'}$ | $+$ | $(\mathbf{G}(x_{\psi\mathbf{R}\psi'} \rightarrow \boxed{x_{\psi'}}))$, $(\mathbf{G}(x_{\psi\mathbf{R}\psi'} \rightarrow (\boxed{x_\psi} \vee \mathbf{X}x_{\psi\mathbf{R}\psi'})))$ |
| | | $-$ | $(\mathbf{G}((\neg x_{\psi\mathbf{R}\psi'}) \rightarrow ((\neg\boxed{x_{\psi'}}) \vee \neg\boxed{x_\psi})))$, $(\mathbf{G}((\neg x_{\psi\mathbf{R}\psi'}) \rightarrow ((\neg\boxed{x_{\psi'}}) \vee \mathbf{X}\neg x_{\psi\mathbf{R}\psi'})))$, $(\mathbf{G}((\neg x_{\psi\mathbf{R}\psi'}) \rightarrow \mathbf{F}\neg\boxed{x_{\psi'}}))$ |
| $\mathbf{F}\psi$ | $x_{\mathbf{F}\psi}$ | $+$ | $(\mathbf{G}(x_{\mathbf{F}\psi} \rightarrow \mathbf{F}\boxed{x_\psi}))$ |
| | | $-$ | $(\mathbf{G}((\neg x_{\mathbf{F}\psi}) \rightarrow \mathbf{X}\neg x_{\mathbf{F}\psi}))$, $(\mathbf{G}((\neg x_{\mathbf{F}\psi}) \rightarrow \neg\boxed{x_\psi}))$ |
| $\mathbf{G}\psi$ | $x_{\mathbf{G}\psi}$ | $+$ | $(\mathbf{G}(x_{\mathbf{G}\psi} \rightarrow \mathbf{X}x_{\mathbf{G}\psi}))$, $(\mathbf{G}(x_{\mathbf{G}\psi} \rightarrow \boxed{x_\psi}))$ |
| | | $-$ | $(\mathbf{G}((\neg x_{\mathbf{G}\psi}) \rightarrow \mathbf{F}\neg\boxed{x_\psi}))$ |

occurrence of a Boolean or temporal operator in $\phi$ one fresh atomic proposition $x, x', \ldots$ is introduced in $SNF(\phi)$ by the translation, and (iii) the size of $SNF(\phi)$ is linear in the size of $\phi$.

As an example we translate the formula $\phi$ shown in (10) into SNF.

$$\phi \equiv (\mathbf{G}(p \wedge q)) \wedge \mathbf{F}\neg p \tag{10}$$

The SNF of $\phi$, $SNF(\phi)$, is given in (11). $x_{(\mathbf{G}(p\wedge q))\wedge\mathbf{F}\neg p}$ represents $(\mathbf{G}(p \wedge q)) \wedge \mathbf{F}\neg p$ in the sense that if in a satisfying assignment for $SNF(\phi)$ $x_{(\mathbf{G}(p\wedge q))\wedge\mathbf{F}\neg p}$ is TRUE at time point $i$, then $(\mathbf{G}(p \wedge q)) \wedge \mathbf{F}\neg p$ is TRUE at time point $i$ on that assignment. The same holds for the other fresh atomic propositions $x_{\mathbf{G}(p\wedge q)}$, $x_{p\wedge q}$, $x_{\mathbf{F}\neg p}$, and $x_{\neg p}$ in $SNF(\phi)$.[6] Together $(x_{(\mathbf{G}(p\wedge q))\wedge\mathbf{F}\neg p})$ and $(\mathbf{G}(x_{(\mathbf{G}(p\wedge q))\wedge\mathbf{F}\neg p} \rightarrow x_{\mathbf{G}(p\wedge q)}))$ force $x_{\mathbf{G}(p\wedge q)}$ to be TRUE at time point 0. Similarly, $x_{\mathbf{F}\neg p}$ is forced to be TRUE at time point 0 via $(\mathbf{G}(x_{(\mathbf{G}(p\wedge q))\wedge\mathbf{F}\neg p} \rightarrow x_{\mathbf{F}\neg p}))$. $\mathbf{G}(p \wedge q)$ is translated into four clauses $(\mathbf{G}(x_{\mathbf{G}(p\wedge q)} \rightarrow \mathbf{X}x_{\mathbf{G}(p\wedge q)}))$, $(\mathbf{G}(x_{\mathbf{G}(p\wedge q)} \rightarrow x_{p\wedge q}))$, $(\mathbf{G}(x_{p\wedge q} \rightarrow p))$, and $(\mathbf{G}(x_{p\wedge q} \rightarrow q))$. With $x_{\mathbf{G}(p\wedge q)}$ being TRUE at time point 0 the first of these four clauses makes $x_{\mathbf{G}(p\wedge q)}$ TRUE at all time points. The remaining three clauses then

[6] Note that in our example fresh atomic propositions $x, x', \ldots$ in $SNF(\phi)$ only represent positive polarity occurrences of subformulas. If there were a negative polarity occurrence of some subformula $\psi$ represented by some fresh atomic proposition $x_\psi$, then $x_\psi$ being FALSE at time point $i$ in a satisfying assignment for $\phi$ would imply $\psi$ being FALSE at time point $i$ on that assignment.

make $p$ and $q$ TRUE continuously. Using the truth of $x_{\mathbf{F}\neg p}$ at time point 0 the two last clauses $(\mathbf{G}(x_{\mathbf{F}\neg p} \rightarrow \mathbf{F}x_{\neg p}))$ and $(\mathbf{G}(x_{\neg p} \rightarrow \neg p))$ ensure that $\neg p$ becomes TRUE eventually. Last but not least notice that — as (10) is obviously unsatisfiable — there can be no such satisfying assignment for $SNF(\phi)$.

$$
\begin{aligned}
SNF(\phi) = \{&(x_{(\mathbf{G}(p\wedge q))\wedge\mathbf{F}\neg p}),\\
&(\mathbf{G}(x_{(\mathbf{G}(p\wedge q))\wedge\mathbf{F}\neg p} \rightarrow x_{\mathbf{G}(p\wedge q)})),\\
&(\mathbf{G}(x_{\mathbf{G}(p\wedge q)} \rightarrow \mathbf{X}x_{\mathbf{G}(p\wedge q)})),\\
&(\mathbf{G}(x_{\mathbf{G}(p\wedge q)} \rightarrow x_{p\wedge q})),\\
&(\mathbf{G}(x_{p\wedge q} \rightarrow p)),\\
&(\mathbf{G}(x_{p\wedge q} \rightarrow q)),\\
&(\mathbf{G}(x_{(\mathbf{G}(p\wedge q))\wedge\mathbf{F}\neg p} \rightarrow x_{\mathbf{F}\neg p})),\\
&(\mathbf{G}(x_{\mathbf{F}\neg p} \rightarrow \mathbf{F}x_{\neg p})),\\
&(\mathbf{G}(x_{\neg p} \rightarrow \neg p))\}.
\end{aligned}
\tag{11}
$$

## 3.4 Temporal Resolution in TRP++

In this subsection we describe temporal resolution (TR) [39] as implemented in TRP++ [53, 54]. We provide a concise description of TR in TRP++ as required for the purposes of this article (Sec. 3.4.1), followed by an example (Sec. 3.4.2). Temporal resolution has been developed since the early 1990s [37], and an extensive body of literature exists. It is out of the scope of this article to provide a detailed introduction or a tutorial on the subject. The following references are among the most suitable as an introduction to TR as needed for this article: [39] provides a general overview of the method and is a good starting point, [32, 33] explains BFS loop search as used in TRP++, and [54] covers the implementation of TR in TRP++. In [82] we provide some intuition on temporal resolution with a slant towards BDD-based symbolic model checking (e.g., [27]).

### *3.4.1 The Temporal Resolution Algorithm in* TRP++

The production rules of TRP++ are shown in Tab. 2. The first column assigns a name to a production rule. The second and fourth columns list the premises. The sixth column gives the conclusion. Columns 3, 5, and 7 are described below.

The algorithm in Fig. 3 provides a high level view of TR in TRP++ [54]. The algorithm takes a set of starting clauses $C$ in SNF as input. It returns *unsat* if $C$ is found to be unsatisfiable (by deriving $\Box$) and *sat* otherwise. Resolution between two initial or two global clauses or between an initial and a global clause is performed by a straightforward extension of propositional resolution (e.g., [5]). This is expressed in the five production rules listed under *saturation* in Tab. 2. Given a set of SNF clauses $C$ we say that one *saturates* $C$, if one applies these production rules to clauses in $C$ until the empty clause $\Box$ has been derived, or until no new clauses are generated. Resolution between a set of initial and global clauses and an eventuality clause with eventuality literal $l$ requires finding a set of global clauses that allows one to infer conditions under which $\mathbf{XG}\neg l$ holds. Such a set of clauses is called a *loop* in $\neg l$. TRP++ implements the BFS approach to loop search [32, 33, 31]. Loop search involves all production rules in Tab. 2 except init-ii, init-in, step-nn, and step-nx.

In line 1 the algorithm in Fig. 3 initializes $M$ with the set of starting clauses and terminates iff one of these is the empty clause. Then, in line 2, it saturates $M$ (terminating iff the empty clause is generated). In line 3 it *augments* $M$ by applying production rule aug1 to

**Table 2** Production rules used in TRP++. Let $P \equiv p_1 \vee \ldots \vee p_n$, $Q \equiv q_1 \vee \ldots \vee q_{n'}$, $R \equiv r_1 \vee \ldots \vee r_{n''}$, and $S \equiv s_1 \vee \ldots \vee s_{n'''}$.

| rule | premise 1 | part. | premise 2 | part. | conclusion | part. |
|---|---|---|---|---|---|---|
| saturation | | | | | | |
| init-ii | $(P \vee l)$ | $M$ | $((\neg l) \vee Q)$ | $M$ | $(P \vee Q)$ | $M$ |
| init-in | $(P \vee l)$ | $M$ | $(\mathbf{G}((\neg l) \vee Q))$ | $M$ | $(P \vee Q)$ | $M$ |
| step-nn | $(\mathbf{G}(P \vee l))$ | $M$ | $(\mathbf{G}((\neg l) \vee Q))$ | $M$ | $(\mathbf{G}(P \vee Q))$ | $M$ |
| step-nx | $(\mathbf{G}(P \vee l))$ | $M$ | $(\mathbf{G}(Q \vee \mathbf{X}((\neg l) \vee R)))$ | $M$ | $(\mathbf{G}(Q \vee \mathbf{X}(P \vee R)))$ | $M$ |
| step-xx | $(\mathbf{G}(P \vee \mathbf{X}(Q \vee l)))$ | $ML$ | $(\mathbf{G}(R \vee \mathbf{X}((\neg l) \vee S)))$ | $ML$ | $(\mathbf{G}(P \vee R \vee \mathbf{X}(Q \vee S)))$ | $ML$ |
| augmentation | | | | | | |
| aug1 | $(\mathbf{G}(P \vee \mathbf{F} l))$ | | | $M$ | $(\mathbf{G}(P \vee l \vee wl))$ | $M$ |
| aug2 | $(\mathbf{G}(P \vee \mathbf{F} l))$ | | | $M$ | $(\mathbf{G}((\neg wl) \vee \mathbf{X}(l \vee wl)))$ | $M$ |
| BFS loop search | | | | | | |
| BFS-loop-it-init-x | $c \equiv (\mathbf{G}(P \vee \mathbf{X}(q_1 \vee \ldots \vee q_{n'})))$ with $n' > 0$ | | | $M$ | $c$ | $L$ |
| BFS-loop-it-init-n | $(\mathbf{G}\ P)$ | | | $M$ | $(\mathbf{G}\,\mathbf{X}\,P)$ | $L$ |
| BFS-loop-it-init-c | $(\mathbf{G}\ P)$ | $L'$ | $(\mathbf{G}(Q \vee \mathbf{F} l))$ | $M$ | $(\mathbf{GX}(P \vee l))$ | $L$ |
| BFS-loop-it-sub | $c \equiv (\mathbf{G}\ P)$ with $c \rightarrow (\mathbf{G}\ Q)$ | | | $L$ | $(\mathbf{GX}(Q \vee l))$ generated by BFS-loop-it-init-c | $L$ |
| BFS-loop-conclusion1 | $(\mathbf{G}\ P)$ | $L$ | $(\mathbf{G}(Q \vee \mathbf{F} l))$ | $M$ | $(\mathbf{G}(P \vee Q \vee l))$ | $M$ |
| BFS-loop-conclusion2 | $(\mathbf{G}\ P)$ | $L$ | $(\mathbf{G}(Q \vee \mathbf{F} l))$ | $M$ | $(\mathbf{G}((\neg wl) \vee \mathbf{X}(P \vee l)))$ | $M$ |

**Input**: A set of SNF clauses $C$.
**Output**: *Unsat* if $C$ is unsatisfiable; *sat* otherwise.

```
1  M ← C; if □ ∈ M then return unsat;
2  saturate(M); if □ ∈ M then return unsat;
3  augment(M);
4  saturate(M); if □ ∈ M then return unsat;
5  M' ← ∅;
6  while M' ≠ M do
7      M' ← M;
8      for c ∈ C . c is an eventuality clause do
9          C' ← {□};
10         repeat
11             initialize-BFS-loop-search-iteration(M, c, C', L);
12             saturate-step-xx(L);
13             C' ← {c' ∈ L | c' has empty X part};
14             C'' ← {(G Q) | (GX(Q ∨ l)) ∈ L generated by [BFS-loop-it-init-c]};
15             found ← subsumes(C', C'');
16         until found or C' = ∅;
17         if found then
18             derive-BFS-loop-search-conclusions(c, C', M);
19             saturate(M); if □ ∈ M then return unsat;
20 return sat;
```

**Fig. 3** LTL satisfiability checking via TR in TRP++

each eventuality clause in $M$ and [aug2] once per eventuality literal in $M$, where $wl$ is a fresh proposition. This is followed by another round of saturation in line 4.[7] From now on the algorithm in Fig. 3 alternates between searching for a loop for some eventuality clause $c$ (lines

[7] Here we report the algorithm as implemented in the version of TRP++ that we obtained. There saturation is performed directly before and directly after augmentation.

9–18) and saturating $M$ if loop search has generated new clauses (line 19). It terminates if either the empty clause was derived (line 19) or if no new clauses were generated (line 20).

Loop search for some eventuality clause $c$ may take several *iterations* (lines 11–15). Each loop search iteration uses saturation restricted to step-xx as a subroutine (line 12). Correctness of BFS loop search requires that each BFS loop search iteration has its own set of clauses $L$ in which it works. We call $M$ and $L$ *partitions*. The set of partitions is a set of sets; i.e., a clause may appear in several partitions but not more than once in any one partition. Columns 3, 5, and 7 in Tab. 2 indicate whether a premise (resp. conclusion) of a production rule is taken from (resp. put into) the main partition ($M$), the loop partition of the current loop search iteration ($L$), the loop partition of the previous loop search iteration ($L'$), or either of $M$ or $L$ as long as premises and conclusion are in the same partition ($ML$). In line 11 partition $L$ of a loop search iteration is initialized by applying production rule BFS-loop-it-init-x once to each global clause with non-empty $\mathbf{X}$ part in $M$, rule BFS-loop-it-init-n once to each global clause with empty $\mathbf{X}$ part in $M$, and rule BFS-loop-it-init-c once to each global clause with empty $\mathbf{X}$ part in the partition of the previous loop search iteration $L'$. Notice that by construction at this point $L$ contains only global clauses with non-empty $\mathbf{X}$ part. Then $L$ is saturated using only rule step-xx (line 12). A loop has been found iff each global clause with empty $\mathbf{X}$ part that was derived in the previous loop search iteration is subsumed by at least one global clause with empty $\mathbf{X}$ part that was derived in the current loop search iteration (lines 13–15). Subsumption between a pair of clauses corresponds to an instance of production rule BFS-loop-it-sub; note, though, that this rule does not produce a new clause but records a relation between two clauses to be used later for extraction of a UC. Loop search for $c$ terminates if either a loop has been found or no clauses with empty $\mathbf{X}$ part were derived (line 16). If a loop has been found, rules BFS-loop-conclusion1 and BFS-loop-conclusion2 are applied once to each global clause with empty $\mathbf{X}$ part that was derived in the current loop search iteration (line 18) to obtain the loop search conclusions for the main partition.

The TR method with a BFS algorithm for loop search, which is implemented in TRP++, is a sound and complete decision procedure for the satisfiability of a set of SNF clauses [53, 54, 39, 32, 33, 31]. We are not aware of a detailed complexity analysis of TR as implemented in TRP++; for complexity analyses of parts of the TR method relevant to the implementation in TRP++ see [39, 33]. To understand the complexity of our extension of the algorithm in Fig. 3 for the extraction of UCs proposed in this article we are mainly interested in the size of the resolution graph that we will construct from an execution of the algorithm in Fig. 3. This will be analyzed in Lemma 1.

### *3.4.2 Example*

We now continue the example from Sec. 3.3. We would like to execute the algorithm in Fig. 3 on the SNF of $\phi \equiv (\mathbf{G}(p \wedge q)) \wedge \mathbf{F}\neg p$. For technical reasons we make some minor modifications. First, when translating $\phi$ into a set of SNF clauses $C$ our implementation treats top level conjuncts as separate formulas. We therefore separately translate $\mathbf{G}(p \wedge q)$ and $\mathbf{F}\neg p$ according to Def. 1. Second, while the use of implication makes the SNF clauses in column 4 of Tab. 1 easier to understand, implication is not an operator that is available for SNF clauses in TRP++. Hence, we syntactically expand implication using its definition. Third, due to space constraints in Fig. 4 we replace the formula indices in the fresh atomic propositions $x, x', \ldots$ with numerical indices. Fourth, we remove some parentheses and let the negation and next time operators bind stronger than the or operator. The modified SNF

of $\phi$, $C$, is shown in (12). $x_1$ corresponds to $x_{\mathbf{G}(p \wedge q)}$, $x_2$ to $x_{p \wedge q}$, $x_3$ to $x_{\mathbf{F}\neg p}$, and $x_4$ to $x_{\neg p}$.

$$C \equiv \begin{array}{l} \{x_1, \\ \quad \mathbf{G}(\neg x_1 \vee \mathbf{X} x_1), \\ \quad \mathbf{G}(\neg x_1 \vee x_2), \\ \quad \mathbf{G}(\neg x_2 \vee p), \\ \quad \mathbf{G}(\neg x_2 \vee q), \\ \quad x_3, \\ \quad \mathbf{G}(\neg x_3 \vee \mathbf{F} x_4), \\ \quad \mathbf{G}(\neg p \vee \neg x_4)\}. \end{array} \tag{12}$$

In Fig. 4 we show an execution of the algorithm in Fig. 3 on $C$. In Fig. 4 TR generally proceeds from bottom to top. At the bottom in the rectangle shaded in light red are the clauses in $C$. The leftmost clause in the top row is the empty clause $\Box$, indicating unsatisfiability. Clauses are connected with directed edges from premises to conclusions. Edges are labeled with production rules, where "BFS-loop" is abbreviated to "loop", "init" to "i", and "conclusion" to "conc". Please ignore the different colors and styles of the clauses and edges for now. These will be explained when discussing extraction of a UC later in Sec. 4.1.

Saturation in line 2 of the algorithm in Fig. 3 produces no new clauses.[8] The two clauses in row 2 are generated by augmentation (line 3). The following saturation (line 4) produces no new clauses. The dark green shaded rectangle is the loop partition for the first loop search iteration. Row 3 contains the clauses obtained by initialization of the BFS loop search iteration (line 11). Row 4 then contains the clauses generated from those in row 3 by saturation restricted to step-xx (line 12). The subsumption test fails in this iteration, as $\neg x_1$ (from $\mathbf{G}(\neg x_1)$) does not subsume $\Box$ (from $\mathbf{G}(\mathbf{X} x_4)$) (lines 13–15). The light green shaded rectangle is the loop partition for the second loop search iteration. Row 5 contains the clauses obtained by initialization and row 6 those obtained from them by restricted saturation. This time the subsumption test succeeds, and the loop search conclusions are shown in row 7 (line 18). Finally, while row 8 contains a "blind alley", row 9 has the derivation of the empty clause $\Box$ via saturation (line 19).

## 4 UC Extraction

In this section we present our method to extract a UC from an execution of the algorithm in Fig. 3. We first show how to extract a UC in SNF (Sec. 4.1). Then we map that UC back to LTL (Sec. 4.2).

### 4.1 Extracting a UC in SNF

In this subsection we describe, given an unsatisfiable set of SNF clauses $C$, how to obtain a subset of $C$, $C^{uc}$, that is by itself unsatisfiable. During the execution of the algorithm in Fig. 3 a resolution graph is built that records which clauses were used to generate other clauses. Then the resolution graph is traversed backwards from the empty clause to find

[8] While it may seem that some clauses are not considered for saturation, this is due to either subsumption of one clause by another (e.g., $\mathbf{G}(\neg w x_4 \vee \mathbf{X}\neg x_1 \vee \mathbf{X} x_4)$ obtained from $\mathbf{G}(\neg w x_4 \vee \mathbf{X} x_4 \vee \mathbf{X} w x_4)$ and $\mathbf{G}(\neg x_1 \vee \neg w x_4)$ is subsumed by $\mathbf{G}(\neg w x_4 \vee \mathbf{X}\neg x_1)$) or the fact that TRP++ uses *ordered* resolution (e.g., $x_1$ with $\mathbf{G}(\neg x_1 \vee x_2)$ — the order here is $x_1 < x_2 < p < q < x_3 < x_4$; [53, 5]). Both are issues of completeness of TR and, therefore, not discussed in this article.

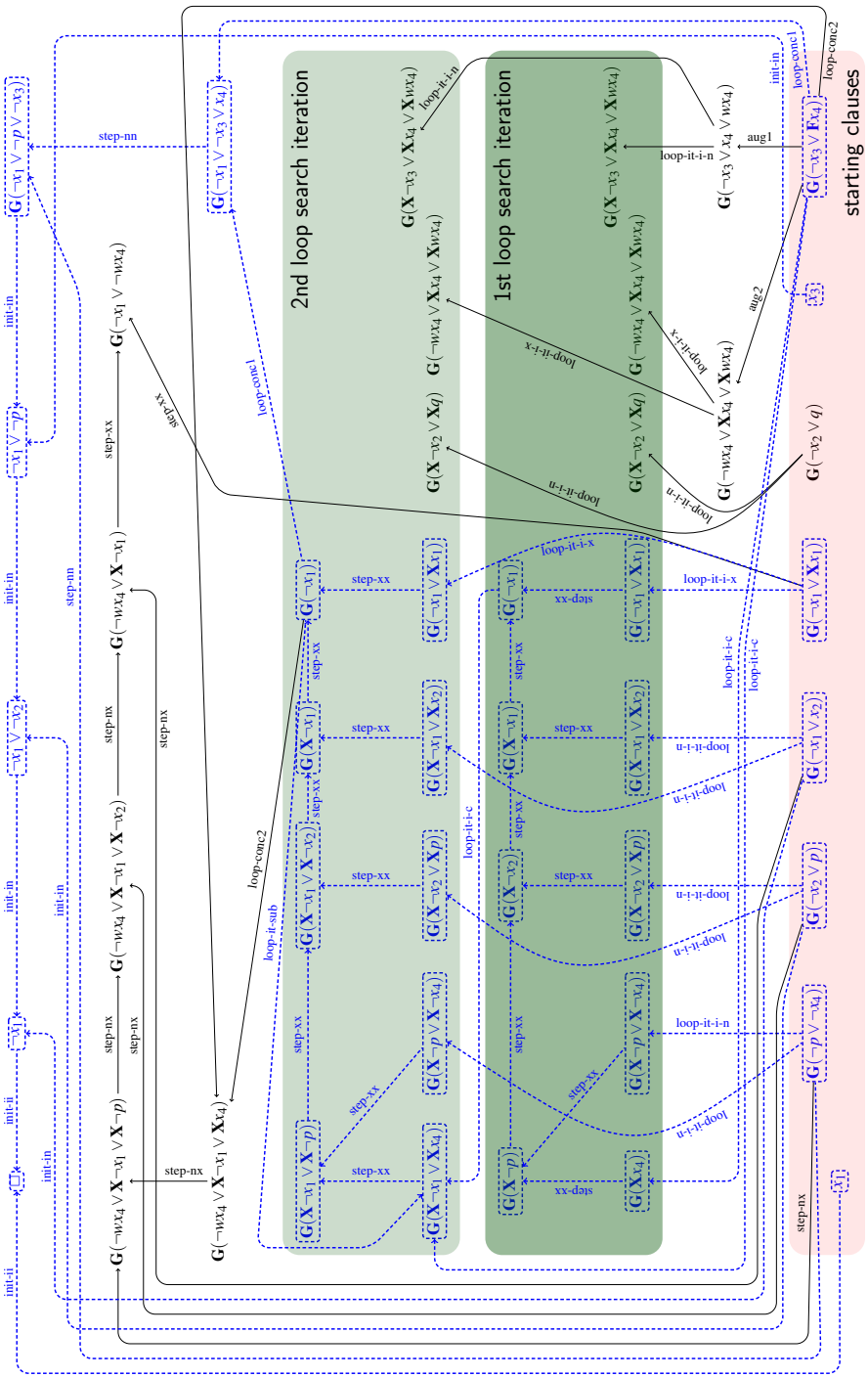



**Fig. 4** Example of an execution of the TR algorithm with corresponding resolution graph and UC extraction in SNF

the subset of $C$ that was actually used to prove unsatisfiability (Sec. 4.1.1). The specifics of the TR algorithm are then used to construct an optimized version of the resolution graph (Sec. 4.1.2).

While we are not aware of corresponding previous work in the domain of temporal logic, the general idea of the construction is unsurprising. In fact, applying a resolution method to a set of input clauses in a clausal normal form, constructing a resolution graph, and determining an unsatisfiable subset of the set of input clauses by backward traversal of the resolution graph from the empty clause is well known in SAT (e.g., [99]). The complexity analysis and the optimization of the resolution graph are original and specific to the algorithm in Fig. 3.

Note that in the preliminary version of this paper [85] we presented the optimized version of the resolution graph right away.

### *4.1.1 Extracting a UC in SNF from a Resolution Graph*

*Resolution Graph — Construction and Added Complexity* In Def. 2–4 we state the construction of a resolution graph during an execution of the algorithm in Fig. 3. Then we discuss the complexity that the construction of a resolution graph adds to an execution of the algorithm in Fig. 3 (Rem. 3 and 4). Finally, in Lemma 1 we establish a bound on the size of a resolution graph.

The definition of a resolution graph is split into three parts. Definition 2 gives the "type" of a resolution graph. Definition 3 describes the initialization of a resolution graph at the beginning of an execution of the algorithm in Fig. 3. Finally, Def. 4 describes how a resolution graph is updated during an execution of the algorithm in Fig. 3. Splitting the definition of a resolution graph allows to limit the scope of the changes that are required for the definition of an optimized resolution graph in Sec. 4.1.2 to how a resolution graph is updated.

**Definition 2 (Resolution Graph: Type)** Let $C$ be a set of SNF clauses, and assume an execution of the algorithm in Fig. 3 on $C$. A *resolution graph* $G$ is a directed graph consisting of (i) a set of vertices $V$, (ii) a set of directed edges $E \subseteq V \times V$, (iii) a labeling of vertices with SNF clauses $L_V : V \to \mathbb{C}$, and (iv) a partitioning $\mathscr{P}^V$ of the set of vertices $V$ into one main partition $M^V$ and one partition $L_i^V$ for each BFS loop search iteration in the execution of the algorithm in Fig. 3 on the set of SNF clauses $C$: $\mathscr{P}^V : V = M^V \uplus L_0^V \uplus \ldots \uplus L_n^V$.

**Definition 3 (Resolution Graph: Initialization)** Let $C$ be a set of SNF clauses, and assume an execution of the algorithm in Fig. 3 on $C$. The *resolution graph $G$ is initialized* in line 1 of the algorithm in Fig. 3 as follows: (i) $V$ contains one vertex $v$ per clause $c$ in $C$: $V = \{v_c \mid c \in C\}$, (ii) $E$ is empty: $E = \emptyset$, (iii) each vertex is labeled with the corresponding clause: $L_V : V \to C, L_V(v_c) = c$, and (iv) the partitioning $\mathscr{P}^V$ contains only the main partition $M^V$, which contains all vertices: $\mathscr{P}^V : M^V = V$.

**Definition 4 (Resolution Graph: Update)** Let $C$ be a set of SNF clauses, and assume an execution of the algorithm in Fig. 3 on $C$. The *resolution graph $G$ is updated* as follows. Whenever a new BFS loop search iteration is entered (line 11), a new partition $L_i^V$ is created and added to $\mathscr{P}^V$. For each application of a production rule from Tab. 2 that either generates a new clause in partition $M^V$ or $L_i^V$ or is the first application of rule BFS-loop-it-sub to clause $c''$ in $C''$ in line 15: (i) if the applied production rule is not rule BFS-loop-it-sub, then a new vertex $v$ is created for the conclusion $c$ (which is a new clause), labeled with $c$, and put into partition $M^V$ or $L_i^V$; (ii) an edge is created from the vertex labeled with premise 1 (resp. premise 2) in partition $M^V$, $L_i^V$, or $L_{i-1}^V$ to the vertex labeled with the conclusion in partition $M^V$ or $L_i^V$.

*Remark 2 (Determinants of Resolution Graph)* Note that a resolution graph constructed according to Def. 2–4 is determined not only by the set of input clauses $C$ but also by the execution of the algorithm in Fig. 3 on $C$. For example, consider $C \equiv \{(p),(\neg p),(q),(\neg q)\}$. Depending on the order in which clauses are considered by the algorithm in Fig. 3 the empty clause $\square$ will be derived either from $(p)$ and $(\neg p)$ or from $(q)$ and $(\neg q)$. As saturation halts as soon as $\square$ has been derived, two different resolution graphs for $C$ will be obtained. Hence, strictly speaking, a resolution graph is parameterized by a set of SNF clauses $C$ and by an execution of the algorithm in Fig. 3 on $C$. In this article both parameters are usually provided by the context. Therefore, in the remainder of this article when we say "resolution graph", then we refer to the object that is obtained from Def. 2–4 after an execution of the algorithm in Fig. 3 on a set of clauses $C$.

Remember that BFS loop search requires that clauses in different BFS loop search iterations are kept apart from each other. Hence, in any partition there can be at most one vertex labeled with a given clause, but there may well exist two or more vertices in different partitions labeled with the same clause. In fact, an application of production rule BFS-loop-it-init-x will lead to such a situation as it copies a clause from the main partition to the partition of the current BFS loop search iteration.

In Rem. 3 and 4 we establish the complexity of the construction of a resolution graph in terms of its own size. In Lemma 1 we then obtain a limit on the size of the resolution graph by bounding the number of (i) different clauses in each partition, (ii) iterations in each loop search by the length of the longest monotonically increasing sequence of Boolean formulas over $AP$, and (iii) loop searches by the number of different loop search conclusions.

*Remark 3 (Vertex in Resolution Graph has At Most Three Incoming Edges)* Let $G$ be a resolution graph. Inspection of Tab. 2 shows that each vertex in the resolution graph $G$ has at most three incoming edges. Let $v_c$ be a vertex in $G$ labeled with clause $c$. If the clause $c$ was generated by an application of any production rule except for BFS-loop-it-init-c, then the vertex $v_c$ only has incoming edges originating at the premises of $c$, and each production rule in Tab. 2 has at most two premises. Otherwise, if the clause $c$ was generated by an application of production rule BFS-loop-it-init-c, then the vertex $v_c$ has at most two incoming edges originating at the premises of $c$ and at most one incoming edge from an application of rule BFS-loop-it-sub.

*Remark 4 (Added Complexity of Construction of Resolution Graph)* Let $G$ be a resolution graph with set of vertices $V$ and set of edges $E$. Assume that the data structure used to represent clauses has an entry for a pointer to the vertex of $G$ that it labels, and similarly the data structure used to represent vertices has an entry for a pointer to the clause that it is labeled with. Moreover, the data structure used to represent vertices has three slots with pointers to its incoming edges (remember that by Rem. 3 each vertex in $G$ has at most three incoming edges) and a Boolean flag to mark vertices in the main partition that are labeled with an initial clause. It is now easy to see that initializing the resolution graph with respect to the set of SNF clauses $C$ and updating the resolution graph $G$ whenever a new clause is generated during the execution of the algorithm in Fig. 3 can be performed using constant time for each clause. In other words, the construction of the resolution graph takes time $\mathcal{O}(|V|+|E|)$ overall in addition to the time required to run the algorithm in Fig. 3.

**Lemma 1 (Size of Resolution Graph)** *Let $C$ be a set of SNF clauses, and let $G$ be a resolution graph with set of vertices $V$ and set of edges $E$. Then $|V|$ and $|E|$ are at most exponential in $|AP|+log(|C|)$.*

*Proof* The following reasoning shows that $|V|$ is at most exponential in $|AP| + log(|C|)$:

1. In an initial clause a proposition can be not present, present non-negated, or present negated. Hence, the number of different initial clauses is $\mathcal{O}(3^{|AP|})$.
2. In a global clause a proposition can be one of not present, present non-negated, or present negated; and prefixed by **X** not present, present non-negated, or present negated. Hence, the number of different global clauses is $\mathcal{O}(9^{|AP|})$.
3. The number of clauses in the main partition is bounded by $|C| + \mathcal{O}(3^{|AP|}) + \mathcal{O}(9^{|AP|}) = \mathcal{O}(|C| + 9^{|AP|})$.
4. The number of clauses in a partition for a BFS loop search iteration is bounded by $\mathcal{O}(9^{|AP|})$.
5. The number of partitions is bounded by 1 plus the number of BFS loop search iterations.
6. The number of iterations in a BFS loop search is bounded by the length of the longest monotonically increasing sequence of Boolean formulas over $AP$, which is $\mathcal{O}(2^{|AP|})$. (In [33] that result is obtained by reasoning on a behavior graph.)
7. The number of BFS loop searches is bounded by the number of different clauses that can be the result of a BFS loop search. The number of different clauses that can be the consequence of BFS loop search conclusion 1 BFS-loop-conclusion1 is bounded by the number of different global clauses with empty next part, which is $\mathcal{O}(3^{|AP|})$. The number of different clauses that can be the consequence of BFS loop search conclusion 2 BFS-loop-conclusion2 is bounded by the number of different eventuality literals times the number of different global clauses with empty next part, which is $\mathcal{O}(|C| \cdot 3^{|AP|})$. Hence, the number of BFS loop searches is bounded by $\mathcal{O}(|C| \cdot 3^{|AP|})$.
8. Taking all of the above into account, the number of clauses is bounded by $\mathcal{O}(|C| + 9^{|AP|} + |C| \cdot 3^{|AP|} \cdot 2^{|AP|} \cdot 9^{|AP|}) = \mathcal{O}(|C| \cdot 54^{|AP|})$.

To see the result for $|E|$ notice that by Rem. 3 each vertex in $G$ has at most three incoming edges. This concludes the proof. □

*UC in SNF — Construction, Correctness, and Added Complexity* We are now ready to describe the extraction of a UC in SNF from a resolution graph in Def. 6. Theorem 1, which establishes correctness of the construction, is then straightforward to obtain.

**Definition 5 (UC in SNF)** Let $C$ be an unsatisfiable set of SNF clauses. Let $C^{uc}$ be an unsatisfiable subset of $C$. Then $C^{uc}$ is a *UC of $C$ in SNF*.

**Definition 6 (UC in SNF via TR)** Let $C$ be an unsatisfiable set of SNF clauses, let $G$ be a resolution graph, and let $v_\Box$ be the (unique) vertex in the main partition $M^V$ of the resolution graph $G$ labeled with the empty clause $\Box$. Let $G'$ be the smallest subgraph of $G$ that contains $v_\Box$ and all vertices in $G$ (and the corresponding edges) that are backward reachable from $v_\Box$. The *UC of $C$ in SNF via TR*, $C^{uc}$, is the subset of $C$ such that there exists a vertex $v$ in the subgraph $G'$, labeled with $c \in C$, and contained in the main partition $M^V$ of $G$: $C^{uc} = \{c \in C \mid \exists v \in V_{G'} \,.\, L_V(v) = c \wedge v \in M^V\}$.

**Theorem 1 (Unsatisfiability of UC in SNF via TR)** *Let $C$ be an unsatisfiable set of SNF clauses, and let $C^{uc}$ be a UC of $C$ in SNF via TR. Then $C^{uc}$ is unsatisfiable.*

*Proof* Notice that in the resolution graph each conclusion is connected by an edge to all of its premises. Therefore, the UC in SNF according to Def. 6 contains all clauses of the set of starting clauses $C$ that contributed to deriving the empty clause and, hence, to establishing unsatisfiability of $C$. It now follows directly from the correctness of TR that $C^{uc}$ is unsatisfiable. This concludes the proof. □

As $C^{uc}$ is a subset of $C$, we have the following corollary.

**Corollary 1 (UC in SNF via TR is UC in SNF)** *Let $C$ be an unsatisfiable set of SNF clauses, and let $C^{uc}$ be a UC of $C$ in SNF via TR. Then $C^{uc}$ is a UC of $C$ in SNF.*

*Remark 5 (Determinants of UC in SNF via TR)* Note that an unsatisfiable set of SNF clauses $C$ may have several UCs in SNF. Moreover, not all of them might be obtained using Def. 6. For example, for $C \equiv \{(p), (\neg p), (\mathbf{G}\, q), (\mathbf{F}\, \neg q)\}$ the algorithm in Fig. 3 will derive the empty clause $\Box$ from $(p)$ and $(\neg p)$ during the first round of saturation, leading to $\{(p), (\neg p)\}$ as a UC of $C$ in SNF via TR. The alternative UC of $C$ in SNF, $\{(\mathbf{G}\, q), (\mathbf{F}\, \neg q)\}$, requires loop search, which is only performed later in the algorithm in Fig. 3 and, therefore, will not be produced by Def. 6. Finally, as Def. 6 relies on a resolution graph, the determinants of a resolution graph in Rem. 2 also become determinants of a UC in SNF via TR.

Next we discuss the complexity that our method for UC extraction adds to an execution of the algorithm in Fig. 3. Using Rem. 4 and Lemma 1 the desired result is easily obtained in Prop. 1. It turns out that the effort that is induced by our method for UC extraction in addition to the effort required by an execution of the algorithm in Fig. 3 is linearly bounded by the effort required by the algorithm in Fig. 3: For all vertices present in the resolution graph a corresponding clause must have been previously generated by the algorithm in Fig. 3, and each vertex in the resolution graph requires constant time for construction and traversal.

**Proposition 1 (Added Complexity of UC Extraction)** *Let $C$ be an unsatisfiable set of SNF clauses. Construction of $C^{uc}$ according to Def. 6 can be performed in time exponential in $|AP| + log(|C|)$ in addition to the time required to run the algorithm in Fig. 3.*

*Proof* Let $G$ be the resolution graph with set of vertices $V$ and set of edges $E$. Assume data structures for clauses and vertices in $G$ are as in Rem. 4. By Rem. 4 construction of $G$ takes time $\mathcal{O}(|V| + |E|)$ overall. Once the empty clause $\Box$ has been derived in the main partition, backward traversal of $G$ from the unique vertex in the main partition labeled with the empty clause, $v_{\Box}$, can be performed (e.g., using breadth first search) in time $\mathcal{O}(|V| + |E|)$. Whenever a vertex labeled with an initial clause $c$ is encountered during the backward traversal of $G$, then $c$ is signaled to be part of $C^{uc}$, requiring time $\mathcal{O}(|C|)$ overall. Using Lemma 1 the result follows. This concludes the proof. $\Box$

*Remark 6 (Pruning the Resolution Graph at Run Time)* The specifics of TR in the algorithm in Fig. 3 allow to optimize extraction of UCs by pruning the resolution graph during the execution of the algorithm in Fig. 3 extended with the construction in Def. 6 as follows. Notice that after the completion of a (successful or unsuccessful) loop search for some eventuality clause $c$ in lines 9–19 of the algorithm in Fig. 3 no new edges between the main partition and one of the partitions used during the just completed loop search for $c$ will be created. Hence, after completion of an execution of lines 9–19 of the algorithm in Fig. 3 vertices in the partitions used during the just completed loop search that are not backward reachable from the main partition can be pruned from the resolution graph.

*Example* We now continue the running example from Sec. 3.4.2. We show how to extract a UC of $\phi$ (see (10)) in SNF via TR from the execution of the algorithm in Fig. 3 on $C$ (see (12)) in Fig. 4, which contains the resolution graph according to Def. 2–4. Vertices are given by the clauses they are labeled with and by the partition in which they appear. We now apply Def. 6. The dashed, blue clauses and edges show the part of the resolution

graph that is backward reachable from $\Box$. Clause $\mathbf{G}(\neg x_2 \vee q)$ is the only clause of $C$ that is not backward reachable from $\Box$. Hence, the UC of $\phi$ in SNF via TR according to Def. 6 is $C^{uc} = C \setminus \{\mathbf{G}(\neg x_2 \vee q)\}$ as shown in (13).

$$C^{uc} \equiv \begin{array}{l} \{x_1, \\ \quad \mathbf{G}(\neg x_1 \vee \mathbf{X} x_1), \\ \quad \mathbf{G}(\neg x_1 \vee x_2), \\ \quad \mathbf{G}(\neg x_2 \vee p), \\ \quad x_3, \\ \quad \mathbf{G}(\neg x_3 \vee \mathbf{F} x_4), \\ \quad \mathbf{G}(\neg p \vee \neg x_4)\}. \end{array} \tag{13}$$

*4.1.2 Extracting a UC in SNF from an Optimized Resolution Graph*

In Def. 7 we optimize the construction of a resolution graph by not including edges between some premises and conclusions. Definition 8 correspondingly adapts the extraction of a UC. The proof of correctness of the optimized construction is then significantly more complex than for the unoptimized variant. Theorem 2 is the main theorem and Lemmas 2–5 contain the details. Proposition 2 complements the definition and proof of correctness of the optimized construction by showing for the remaining edges that they are indeed required to obtain a UC.

The "type" and initialization of an optimized resolution graph are as for a resolution graph (Def. 2, 3); only the update changes (Def. 7). In the remainder of this article when we say "optimized resolution graph", then we refer to the object that is obtained from Def. 2, 3, and 7 after an execution of the algorithm in Fig. 3 on a set of clauses $C$.

**Definition 7 (Optimized Resolution Graph: Update)** An *optimized resolution graph is updated* in the same way as a resolution graph, except that contrary to Def. 4 no edge is added between a pair of vertices labeled with a premise and a conclusion in the following four cases: (i) between vertices labeled with premise 1 and the conclusion of rule aug2, (ii) between vertices labeled with premise 1 and the conclusion of rule BFS-loop-it-init-c, (iii) between vertices labeled with premise 2 and the conclusion of rule BFS-loop-it-init-c, and (iv) between vertices labeled with premise 2 and the conclusion of rule BFS-loop-conclusion2.

**Definition 8 (UC in SNF via TR from an Optimized Resolution Graph)** Let $C$ be an unsatisfiable set of SNF clauses. The *UC of $C$ in SNF via TR from an optimized resolution graph* is obtained in the same way as the UC of $C$ in SNF via TR, except that an optimized resolution graph is used.

**Theorem 2 (Unsatisfiability of UC in SNF via TR from an Optimized Resolution Graph)** *Let $C$ be an unsatisfiable set of SNF clauses, and let $C^{uc}$ be a UC of $C$ in SNF via TR from an optimized resolution graph. Then $C^{uc}$ is unsatisfiable.*

*Proof* Assume for a moment that in Def. 7 no edges are excluded in the update of the optimized resolution graph. In that case unsatisfiability of $C^{uc}$ has been established in Thm. 1. In the remainder of the proof we show that for constructing a UC of $C$ in SNF via TR the optimized resolution graph can be used in place of the resolution graph, i.e., that none of the four exclusions of edges in Def. 7 render the resulting UC of $C$ in SNF satisfiable.

We have to show that (i) not including an edge from the vertex labeled with premise 1 to the vertex labeled with the conclusion for an application of rule aug2, (ii) not including

an edge from the vertex labeled with premise 2 to the vertex labeled with the conclusion for an application of rule BFS-loop-conclusion2, (iii) not including an edge from the vertex labeled with premise 2 to the vertex labeled with the conclusion for an application of rule BFS-loop-it-init-c, and (iv) not including an edge from the vertex labeled with premise 1 to the vertex labeled with the conclusion for an application of rule BFS-loop-it-init-c in the resolution graph $G$ maintains the fact that the resulting $C^{uc}$ is unsatisfiable.

To see the intuition behind (i) note that for a vertex $v_c$ labeled with the conclusion $c$ of an application of rule aug2 in the main partition $M^V$ to be backward reachable from the (unique) vertex in the main partition $M^V$ of the resolution graph $G$ labeled with the empty clause $\Box$, $v_\Box$, the occurrence of $\neg wl$ in $c$ must be "resolved away" at some point on the path from $v_c$ to $v_\Box$. It turns out that this can only happen by resolution with a clause that is derived from the conclusion of rule aug1 applied to an eventuality clause $c'$ with eventuality literal $l$. By the construction of the resolution graph $G$ $v_{c'}$ must be backward reachable from $v_\Box$ and, therefore, $c'$ must be included in the UC in SNF. Hence, an execution of the algorithm in Fig. 3 with input $C^{uc}$ will produce $c$ from $c'$. For a formal proof see Lemma 2.

A similar reasoning as for (i) applies to (ii), formalized in Lemma 3.

For (iii) notice that a vertex labeled with the conclusion of an application of rule BFS-loop-it-init-c can only be backward reachable from $v_\Box$ if the corresponding BFS loop search iteration is successful and a vertex labeled with one of the resulting conclusions of rules BFS-loop-conclusion1 or BFS-loop-conclusion2 is backward reachable from $v_\Box$. The latter fact implies that an eventuality clause with the same eventuality literal as in premise 2 of rule BFS-loop-it-init-c is present in the UC in SNF. Hence, an execution of the algorithm in Fig. 3 with input $C^{uc}$ will produce premise 2 of BFS-loop-it-init-c as required. This is formally proven in Lemma 4.

Finally, (iv) corresponds to considering only the last iteration of a successful loop search to obtain the UC $C^{uc}$. (iv) is obtained by understanding that in a BFS loop search iteration the premises 1 of rule BFS-loop-it-init-c essentially constitute a hypothetical fixed point; if the BFS loop search iteration is successful, then the hypothetical fixed point is proven to be an actual fixed point. For the correctness of a proof of the unsatisfiability of $C$ it is only relevant that this hypothetical fixed point is shown to be an actual fixed point but not how the hypothesis is obtained. This is formalized in Lemma 5. This concludes the proof. $\Box$

**Lemma 2** *Let $C$ be an unsatisfiable set of SNF clauses, let $G$ be an optimized resolution graph, and let $G'$ be the subgraph according to Def. 8. Let $v_0$ be a vertex in $G'$ labeled with a clause $c_0 = (\mathbf{G}((\neg wl) \vee \mathbf{X}(l \vee wl)))$ created by augmentation aug2 from some eventuality clause $(\mathbf{G}(p_1 \vee \ldots \vee p_n \vee \mathbf{F}l)) \in C$ with eventuality literal $l$. Then there is a vertex $v_1$ in $G'$ labeled with an eventuality clause $c_1 = (\mathbf{G}(q_1 \vee \ldots \vee q_{n'} \vee \mathbf{F}l)) \in C$ with eventuality literal $l$.*

*Proof* There exists a path $\pi$ of non-zero length in $G'$ from $v_0$ to the unique vertex $v_\Box$ in the main partition $M$ labeled with the empty clause $\Box$. On the path $\pi$ there exist two vertices $v_2, v_3$ such that $v_2$ is labeled with a clause $c_2$ that contains $\neg wl$ or $\mathbf{X}\neg wl$, while $v_3$ and all of its successors on $\pi$ are labeled with clauses that contain neither $\neg wl$ nor $\mathbf{X}\neg wl$. Let $c_3$ be the clause labeling $v_3$.

- *Case 1.* $c_3$ is generated by initial or step resolution init-ii, init-in, step-nn, step-nx, or step-xx from $c_2$ and some other clause $c_4$. $c_4$ must contain $wl$ or $\mathbf{X}wl$. Moreover, there must be a path $\pi'$ (possibly of zero length) that starts from a vertex $v_5$ labeled with a clause $c_5$ and that ends in the vertex $v_4$ labeled with $c_4$, such that each vertex on the path $\pi'$ is labeled with a clause that contains $wl$ or $\mathbf{X}wl$. Finally, $wl$ or $\mathbf{X}wl$ must be present in

$c_5$ either because $c_5$ is contained in the set of input clauses in SNF, $C$, or because $c_5$ is generated by some production rule that introduces $wl$ or $\mathbf{X}wl$ in the conclusion.

- *Case 1.1.* $c_5$ is contained in the set of input clauses in SNF, $C$. Impossible: $wl$ is a fresh proposition in aug1 and aug2.
- *Case 1.2.* $c_5$ is generated by initial or step resolution init-ii, init-in, step-nn, step-nx, or step-xx. Impossible: initial and step resolution do not generate literals that are not contained (modulo time-shifting) in at least one of the premises.
- *Case 1.3.* $c_5$ is generated by augmentation 1 aug1. By the construction of the resolution graph $G$ and the subgraph $G'$ there is an edge in $G'$ from a vertex $v_1$ in $G'$ labeled with an eventuality clause $c_1 = (\mathbf{G}(q_1 \vee \ldots \vee q_{n'} \vee \mathbf{F}l)) \in C$ with eventuality literal $l$ to $v_5$.
- *Case 1.4.* $c_5$ is generated by augmentation 2 aug2, i.e., $c_5 = c_0$. This introduces another occurrence of $\neg wl$ to be "resolved away". Note that in the main partition only new clauses are generated from existing ones with edges leading from existing vertices labeled with existing clauses to new vertices labeled with new clauses. Therefore, the main partition of $G'$ is a finite directed acyclic graph, and this case cannot happen infinitely often.
- *Case 1.5.* $c_5$ is generated by BFS loop search initialization BFS-loop-it-init-x. Impossible: the production rule BFS-loop-it-init-x copies a clause verbatim. I.e., it cannot be the case that $c_5$ contains $wl$ or $\mathbf{X}wl$, while the premise does not.
- *Case 1.6.* $c_5$ is generated by BFS loop search initialization BFS-loop-it-init-n. Impossible: the production rule BFS-loop-it-init-n copies and time-shifts a clause. I.e., it cannot be the case that $c_5$ contains $\mathbf{X}wl$, while the premise does not contain $wl$.
- *Case 1.7.* $c_5$ is generated by BFS loop search initialization BFS-loop-it-init-c. Impossible: the production rule BFS-loop-it-init-c copies and time-shifts a clause from a previous BFS loop search iteration (or initializes with the empty clause $\square$) and disjoins with an eventuality literal $\mathbf{X}l'$. I.e., it cannot be the case that $c_5$ contains $\mathbf{X}wl$, while the premise does not contain $wl$.
- *Case 1.8.* $v_5$ is linked to via BFS loop search subsumption BFS-loop-it-sub. This case can be ignored as BFS loop search subsumption BFS-loop-it-sub does not actually generate a clause but merely links existing ones.
- *Case 1.9.* $c_5$ is generated by BFS loop search conclusion 1 BFS-loop-conclusion1. Impossible: production rule BFS-loop-conclusion1 copies all literals verbatim from a clause derived in loop search, copies all literals verbatim from an eventuality clause except for the eventuality literal $l'$ prefixed by $\mathbf{F}$, and disjoins with the eventuality literal $l'$. I.e., it cannot be the case that $c_5$ contains $wl$, while the premises do not.
- *Case 1.10.* $c_5$ is generated by BFS loop search conclusion 2 BFS-loop-conclusion2. Impossible: production rule BFS-loop-conclusion2 copies and time-shifts all literals from a clause $c_6$ derived in loop search and disjoins with $\neg wl'$ and $\mathbf{X}l'$ for some eventuality literal $l'$. I.e., it cannot be the case that $c_5$ contains $\mathbf{X}wl$, while the premise $c_6$ does not contain $wl$.

- *Case 2.* $c_3$ is generated by augmentation aug1 or aug2. Impossible: the premise of the production rules aug1 and aug2 cannot contain either $\neg wl$ or $\mathbf{X}\neg wl$, as $wl$ is assumed to be a fresh proposition in aug1 and aug2.
- *Case 3.* $c_3$ is generated by BFS loop search initialization BFS-loop-it-init-x. Impossible: the production rule BFS-loop-it-init-x copies a clause verbatim. I.e., it cannot be the case that $c_2$ contains $\neg wl$ or $\mathbf{X}\neg wl$, while $c_3$ does not.

- *Case 4.* $c_3$ is generated by BFS loop search initialization BFS-loop-it-init-n. Impossible: the production rule BFS-loop-it-init-n copies and time-shifts a clause. I.e., it cannot be the case that $c_2$ contains $\neg wl$, while $c_3$ does not contain $\mathbf{X}\neg wl$.
- *Case 5.* $c_3$ is generated by BFS loop search initialization BFS-loop-it-init-c. Impossible: the production rule BFS-loop-it-init-c copies and time-shifts a clause from a previous BFS loop search iteration (or initializes with the empty clause $\Box$) and disjoins with an eventuality literal $\mathbf{X}l'$. I.e., it cannot be the case that $c_2$ contains $\neg wl$, while $c_3$ does not contain $\mathbf{X}\neg wl$.
- *Case 6.* $v_2$ and $v_3$ are linked via BFS loop search subsumption BFS-loop-it-sub, i.e., a time-shifted version of $c_2$ subsumes $c_3$. Impossible: BFS-loop-it-sub links from a clause with fewer literals to a clause with (modulo time-shifting) the same and more literals. I.e., it cannot be the case that $c_2$ contains $\neg wl$, while $c_3$ does not contain $\mathbf{X}\neg wl$.
- *Case 7.* $c_3$ is generated by BFS loop search conclusion 1 BFS-loop-conclusion1. Impossible: production rule BFS-loop-conclusion1 copies all literals verbatim from a clause derived in loop search, copies all literals verbatim from an eventuality clause except for the eventuality literal $l'$ prefixed by $\mathbf{F}$, and disjoins with the eventuality literal $l'$. I.e., it cannot be the case that $c_2$ contains $\neg wl$, while $c_3$ does not.
- *Case 8.* $c_3$ is generated by BFS loop search conclusion 2 BFS-loop-conclusion2. Impossible: production rule BFS-loop-conclusion2 copies and time-shifts all literals from a clause derived in loop search and disjoins with $\neg wl'$ and $\mathbf{X}l'$ for some eventuality literal $l'$. I.e., it cannot be the case that $c_2$ contains $\neg wl$, while $c_3$ does not contain $\mathbf{X}\neg wl$.

Notice that the only possible cases are case 1.3 and 1.4. Of those, case 1.4 can only happen a finite number of times and must be followed by an occurrence of case 1.3. This concludes the proof. $\Box$

**Lemma 3** *Let $C$ be an unsatisfiable set of SNF clauses, let $G$ be an optimized resolution graph, and let $G'$ be the subgraph according to Def. 8. Let $v$ be a vertex in $G'$ labeled with a clause $c = (\mathbf{G}((\neg wl) \vee \mathbf{X}(q_1 \vee \ldots \vee q_{n'} \vee l)))$ generated by BFS loop search conclusion 2 BFS-loop-conclusion2 from some eventuality clause $(\mathbf{G}(p_1 \vee \ldots \vee p_n \vee \mathbf{F}l)) \in C$ with eventuality literal $l$ (and some other clause). Then there is a vertex $v''$ in $G'$ labeled with an eventuality clause $c'' = (\mathbf{G}(r_1 \vee \ldots \vee r_{n''} \vee \mathbf{F}l)) \in C$ with eventuality literal $l$.*

*Proof* Analogous to the proof of Lemma 2. $\Box$

**Lemma 4** *Let $C$ be an unsatisfiable set of SNF clauses, let $G$ be an optimized resolution graph, and let $G'$ be the subgraph according to Def. 8. Let $v$ be a vertex in $G'$ labeled with a clause $c = (\mathbf{GX}(q_1 \vee \ldots \vee q_{n'} \vee l))$ generated by production rule BFS-loop-it-init-c from some eventuality clause $(\mathbf{G}(p_1 \vee \ldots \vee p_n \vee \mathbf{F}l)) \in C$ with eventuality literal $l$ (and some other clause). Then there is a vertex $v''$ in $G'$ labeled with an eventuality clause $c'' = (\mathbf{G}(r_1 \vee \ldots \vee r_{n''} \vee \mathbf{F}l)) \in C$ with eventuality literal $l$.*

*Proof* By the construction of the optimized resolution graph $G$ (Def. 2, 3, 7) and its subgraph $G'$ (Def. 8) $v$ is included in $G'$ only if $G'$ also includes some vertex $v'$ labeled with some clause $c'$ such that $c'$ was generated by BFS loop search conclusion BFS-loop-conclusion1 or BFS-loop-conclusion2 from the BFS loop search iteration of which $c$ is part.

- *Case 1.* $c'$ is generated by BFS loop search conclusion 1 BFS-loop-conclusion1. The claim follows from the construction of the optimized resolution graph $G$ and its subgraph $G'$. By Def. 7 $v'$ has an incoming edge from a vertex $v''$ labeled with an eventuality clause $c'' = (\mathbf{G}(r_1 \vee \ldots \vee r_{n''} \vee \mathbf{F}l)) \in C$ with eventuality literal $l$ and by Def. 8 $v''$ is included in $G'$ if $v'$ is included.

– *Case 2.* $c'$ is generated by BFS loop search conclusion 2 BFS-loop-conclusion2. In that case the claim follows directly from Lemma 3.

This concludes the proof. □

**Lemma 5** *Let $C$ be a set of SNF clauses, assume an execution of the algorithm in Fig. 3 on $C$, and let $C' \equiv \{(\mathbf{G}(q_{i',1} \vee \ldots \vee q_{i',n'_{i'}})) \mid 1 \leq i' \leq n'\}$ and $C'' \equiv \{(\mathbf{GX}(p_{i,1} \vee \ldots \vee p_{i,n_i} \vee l)) \mid 1 \leq i \leq n\}$ be the sets of clauses obtained in line 13 and line 14 of the algorithm in Fig. 3 in the last iteration of a successful loop search for eventuality literal $l$. Then, assuming $C$, $\{(\mathbf{G}(q_{i',1} \vee \ldots \vee q_{i',n'_1} \vee \mathbf{XG}\neg l)) \mid 1 \leq i' \leq n'\}$ is provable.*

*Proof* After initialization of a BFS loop search iteration in line 11 of the algorithm in Fig. 3 there are three sets of clauses according to the three production rules for initializing a BFS loop search iteration. Clauses generated by BFS-loop-it-init-x and BFS-loop-it-init-n are (partly time-shifted) duplicates of clauses derived so far in the main partition. BFS-loop-it-init-c generates the set of clauses $C''$. From these three sets saturation restricted to rule step-xx in line 12 derives another set of clauses, $C'$. Taking the restriction of saturation to rule step-xx into account, that BFS loop search iteration has established that, assuming $C$, the following fact is provable:

$$\mathbf{G}((\bigwedge_{1\leq i\leq n} \mathbf{X}(p_{i,1} \vee \ldots \vee p_{i,n_i} \vee l)) \rightarrow \bigwedge_{1\leq i'\leq n'} (q_{i',1} \vee \ldots \vee q_{i',n'_{i'}})). \tag{14}$$

Moreover, because for a successful BFS loop search iteration subsumption in line 15 succeeds, the following fact is also provable:

$$\bigwedge_{1\leq i\leq n} \bigvee_{1\leq i'\leq n'} \mathbf{G}((q_{i',1} \vee \ldots \vee q_{i',n'_{i'}}) \rightarrow (p_{i,1} \vee \ldots \vee p_{i,n_i})). \tag{15}$$

We rewrite (14) and (15) as follows:

$$\begin{aligned}
&\mathbf{G}((\bigwedge_{1\leq i\leq n} \mathbf{X}(p_{i,1} \vee \ldots \vee p_{i,n_i} \vee l)) \rightarrow \bigwedge_{1\leq i'\leq n'} (q_{i',1} \vee \ldots \vee q_{i',n'_{i'}})) \\
\Leftrightarrow\ &\mathbf{G} \bigwedge_{1\leq i'\leq n'} ((\bigwedge_{1\leq i\leq n} \mathbf{X}(p_{i,1} \vee \ldots \vee p_{i,n_i} \vee l)) \rightarrow (q_{i',1} \vee \ldots \vee q_{i',n'_{i'}})) \\
\Leftrightarrow\ &\bigwedge_{1\leq i'\leq n'} \mathbf{G}((\bigwedge_{1\leq i\leq n} \mathbf{X}(p_{i,1} \vee \ldots \vee p_{i,n_i} \vee l)) \rightarrow (q_{i',1} \vee \ldots \vee q_{i',n'_{i'}})) \\
\Leftrightarrow\ &\bigwedge_{1\leq i'\leq n'} \mathbf{G}((\neg(q_{i',1} \vee \ldots \vee q_{i',n'_{i'}})) \rightarrow \neg \bigwedge_{1\leq i\leq n} \mathbf{X}(p_{i,1} \vee \ldots \vee p_{i,n_i} \vee l)) \\
\Leftrightarrow\ &\bigwedge_{1\leq i'\leq n'} \mathbf{G}((\neg(q_{i',1} \vee \ldots \vee q_{i',n'_{i'}})) \rightarrow \bigvee_{1\leq i\leq n} \mathbf{X}\neg(p_{i,1} \vee \ldots \vee p_{i,n_i} \vee l)) \\
\Leftrightarrow\ &\bigwedge_{1\leq i'\leq n'} \mathbf{G}((\neg(q_{i',1} \vee \ldots \vee q_{i',n'_{i'}})) \rightarrow \bigvee_{1\leq i\leq n} \mathbf{X}((\neg(p_{i,1} \vee \ldots \vee p_{i,n_i})) \wedge \neg l)) \\
\Leftrightarrow\ &\bigwedge_{1\leq i'\leq n'} \mathbf{G}((\neg(q_{i',1} \vee \ldots \vee q_{i',n'_{i'}})) \rightarrow ((\mathbf{X}\neg l) \wedge \bigvee_{1\leq i\leq n} \mathbf{X}\neg(p_{i,1} \vee \ldots \vee p_{i,n_i}))),
\end{aligned} \tag{16}$$

$$\begin{aligned}
&\bigwedge_{1\leq i\leq n} \bigvee_{1\leq i'\leq n'} \mathbf{G}((q_{i',1} \vee \ldots \vee q_{i',n'_{i'}}) \rightarrow (p_{i,1} \vee \ldots \vee p_{i,n_i})) \\
\Leftrightarrow\ &\bigwedge_{1\leq i\leq n} \bigvee_{1\leq i'\leq n'} \mathbf{G}((\neg(p_{i,1} \vee \ldots \vee p_{i,n_i})) \rightarrow \neg(q_{i',1} \vee \ldots \vee q_{i',n'_{i'}})).
\end{aligned} \tag{17}$$

Putting (16) and (17) together, we obtain (18), which is exactly the premise required to perform eventuality resolution with an eventuality clause with eventuality literal $l$ [39]:

$$\bigwedge_{1 \leq i' \leq n'} \mathbf{G}(q_{i',1} \vee \ldots \vee q_{i',n'_{i'}} \vee \mathbf{XG}\neg l). \tag{18}$$

This concludes the proof. □

**Corollary 2 (UC in SNF via TR from an Optimized Resolution Graph is UC in SNF)** *Let $C$ be an unsatisfiable set of SNF clauses, and let $C^{uc}$ be a UC of $C$ in SNF via TR from an optimized resolution graph. Then $C^{uc}$ is a UC of $C$ in SNF.*

Theorem 2 shows that not including some premises during the update of the optimized resolution graph still leads to a UC. It does not discuss whether the remaining premises actually need to be included to guarantee a UC. For all premises of all production rules not excluded in the update of the optimized resolution graph in Def. 7 it turns out that they are indeed required to obtain a UC. In other words, for any of the premises not excluded in the update of the optimized resolution graph in Def. 7 there exists a set of SNF clauses $C$ such that excluding that particular premise from the update of the optimized resolution graph for $C$ in Def. 7 and from the subsequent extraction of a UC in SNF in Def. 8 leads to a satisfiable rather than unsatisfiable set of clauses $C^{uc} \subseteq C$.

**Proposition 2 (Minimality of Set of Premises to Include in Optimized Resolution Graph)** *The set of premises included in the update of the optimized resolution graph in Def. 7 is minimal.*

*Proof* The proof is obtained by providing suitable examples. For a given premise $p$ of some production rule $\boxed{\text{rule}}$ with conclusion $c$ the need for inclusion of edges between instances of $p$ and $c$ induced by $\boxed{\text{rule}}$ in the optimized resolution graph to obtain a UC can be established as follows.[9] Let $C$ be an unsatisfiable set of SNF clauses, let $G$ be an optimized resolution graph, let $v_\Box$ be the (unique) vertex in the main partition $M^V$ of the optimized resolution graph $G$ labeled with the empty clause $\Box$, and let $C^{uc}$ be the UC of $C$ in SNF via TR from an optimized resolution graph. Assume that $C^{uc}$ is a minimal UC, i.e., for any $c' \in C^{uc}$ we have that $C^{uc} \setminus \{c'\}$ is satisfiable. Now, if removing all edges between instances of $p$ and $c$ induced by $\boxed{\text{rule}}$ from $G$ makes a vertex $v_{c'}$ in $M^V$ labeled with $c' \in C^{uc}$ backward unreachable from $v_\Box$, then the UC obtained without including edges between instances of $p$ and $c$ induced by $\boxed{\text{rule}}$ clearly would not be unsatisfiable.

Hence, for all premises of all production rules not excluded in the update of the optimized resolution graph in Def. 7 we need to provide triples of premises $p$ of production rules $\boxed{\text{rule}}$, minimally unsatisfiable SNFs $C \equiv C^{uc}$, and subgraphs $G'$ of optimized resolution graphs with the following properties. (i) $G'$ is the subgraph according to Def. 8 for $C$. (ii) There exists a vertex $v_{c'}$ in $G'$ labeled with $c' \in C$ and an edge $e$ that is an instance of $p$ and $c$ induced by $\boxed{\text{rule}}$ such that removal of $e$ from $G'$ makes $v_{c'}$ backward unreachable from $v_\Box$. In the graphs below $e$ and $v_{c'}$ are marked dashed, blue. The vertex labels use `TRP++` syntax.

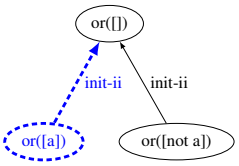
[9] Note that this proof assumes that there are no edges between instances of the premise of $\boxed{\text{aug2}}$, the premises of $\boxed{\text{BFS-loop-it-init-c}}$, and premise 2 of $\boxed{\text{BFS-loop-conclusion2}}$ and their conclusions induced by these production rules as indicated in Def. 7. I.e., different minimal sets of premises to include in an optimized resolution graph may exist.

Note that the search for such triples can be supported by a corresponding modification to the temporal resolution solver. For the more complex cases candidates were obtained in that way and then optimized by hand.

Below we provide triples for init-ii and BFS-loop-it-init-x. The triples for init-in, step-nn, step-nx, step-xx are trivially obtained as the one for init-ii. The triple for BFS-loop-it-init-x also serves as triple for premises 1 and 2 of BFS-loop-conclusion1. The remaining triples are more complex. They can be found in App. A of [86].

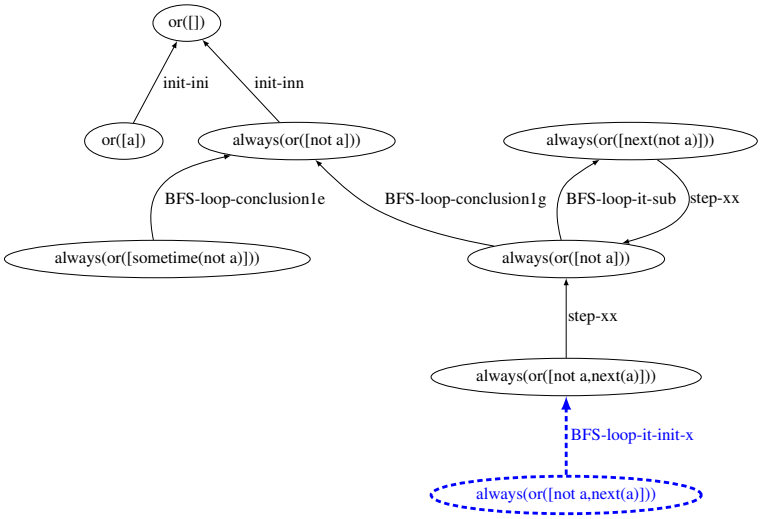
---

init-ii $\{(a), (\neg a)\}$




---

BFS-loop-it-init-x $\{(a), (\mathbf{G}((\neg a) \vee \mathbf{X}a)), (\mathbf{GF}(\neg a))\}$




---

This concludes the proof. □

Notice that using optimized resolution graphs has no impact on the added complexity of UC extraction. Hence, we state Prop. 3.

**Proposition 3 (Added Complexity of UC Extraction from an Optimized Resolution Graph)** *Let $C$ be an unsatisfiable set of SNF clauses. Construction of $C^{uc}$ according to Def. 8 can be performed in time exponential in $|AP| + log(|C|)$ in addition to the time required to run the algorithm in Fig. 3.*

*Remark 7 (Pruning the Optimized Resolution Graph at Run Time)* The fact that not all premises need to be included during the update of the optimized resolution graph permits the following optimization for an optimized resolution graph during the execution of the algorithm in Fig. 3 extended with the construction in Def. 8. Note that, because in the optimized resolution graph there is no edge from a vertex labeled with premise 1 to a vertex labeled with the conclusion of an application of production rule BFS-loop-it-init-c, there are no outgoing edges from a failed loop search iteration (lines 11–15 of the algorithm in Fig. 3). Therefore, if a loop search iteration fails, all vertices and edges in the partition of that loop search iteration can be pruned from the optimized resolution graph right away. Moreover, Rem. 6 also applies to optimized resolution graphs.

In the next Subsec. 4.2 and in the following Sec. 5 we only consider optimized resolution graphs and UCs in SNF via TR from optimized resolution graphs, and we drop the designators "optimized" and "from an optimized resolution graph".

*Example* In Fig. 5 we return to our running example from Sec. 3.4.2 and 4.1.1 to illustrate the construction of the optimized resolution graph and its use to extract a UC in SNF via TR. The graph in Fig. 5 shows the same, unoptimized resolution graph as in Fig. 4. The edges that are excluded from the optimized resolution graph according to Def. 7 are marked dotted, red. The reader can easily verify that the dashed, blue edges and the solid, black edges in Fig. 5 are not excluded in the update of the optimized resolution graph in Def. 7, while the dotted, red edges are in fact excluded there. As in Fig. 4 the dashed, blue clauses and edges show the part of the resolution graph that is backward reachable from the empty clause $\square$. Notice that now no clause in the loop partition for the first loop search iteration (in the rectangle in the middle shaded in dark green), which was unsuccessful, is backward reachable from $\square$. Still, the subset of the starting clauses (in the rectangle at the bottom shaded in light red) that is backward reachable from $\square$ is the same as when using the unoptimized resolution graph in Fig. 4. Therefore, also the UC in SNF via TR from an optimized resolution graph according to Def. 8 is unchanged (see (13)).

## 4.2 Extracting a UC in LTL

In Def. 10 we describe how to map a UC in SNF back to a UC in LTL. The correctness of the construction is then proved in Thm. 3. The main idea in the proof is to compare the SNF of $\phi$ and of its UC in LTL by partitioning the SNF clauses into three sets: one that is shared by the two SNFs, one that replaces some occurrences of propositions in $SNF(\phi)$ with TRUE or FALSE, and one whose clauses are only in $SNF(\phi)$. Then one can show that the UC of $\phi$ in SNF must be contained in the first partition.

**Definition 9 (UC in LTL)** (cf. Def. 10 of [83]) Let $\phi$ be an unsatisfiable LTL formula. Let $\phi^{uc}$ (i) be obtained from $\phi$ by replacing a set of positive polarity occurrences of subformulas of $\phi$ with TRUE and a set of negative polarity occurrences of subformulas of $\phi$ with FALSE and (ii) be unsatisfiable. Then $\phi^{uc}$ is a *UC of $\phi$ in LTL*.

**Definition 10 (UC in LTL from SNF)** Let $\phi$ be an unsatisfiable LTL formula, let $SNF(\phi)$ be its SNF, and let $C^{uc}$ be a UC of $SNF(\phi)$ in SNF. Then the *UC of $\phi$ in LTL from SNF*, $\phi^{uc}$, is obtained as follows. For each positive (resp. negative) polarity occurrence of a proper subformula $\psi$ of $\phi$ with proposition $x_\psi$ according to Tab. 1, replace $\psi$ in $\phi$ with TRUE (resp. FALSE) iff $C^{uc}$ contains no clause with an occurrence of proposition $x_\psi$ that is marked

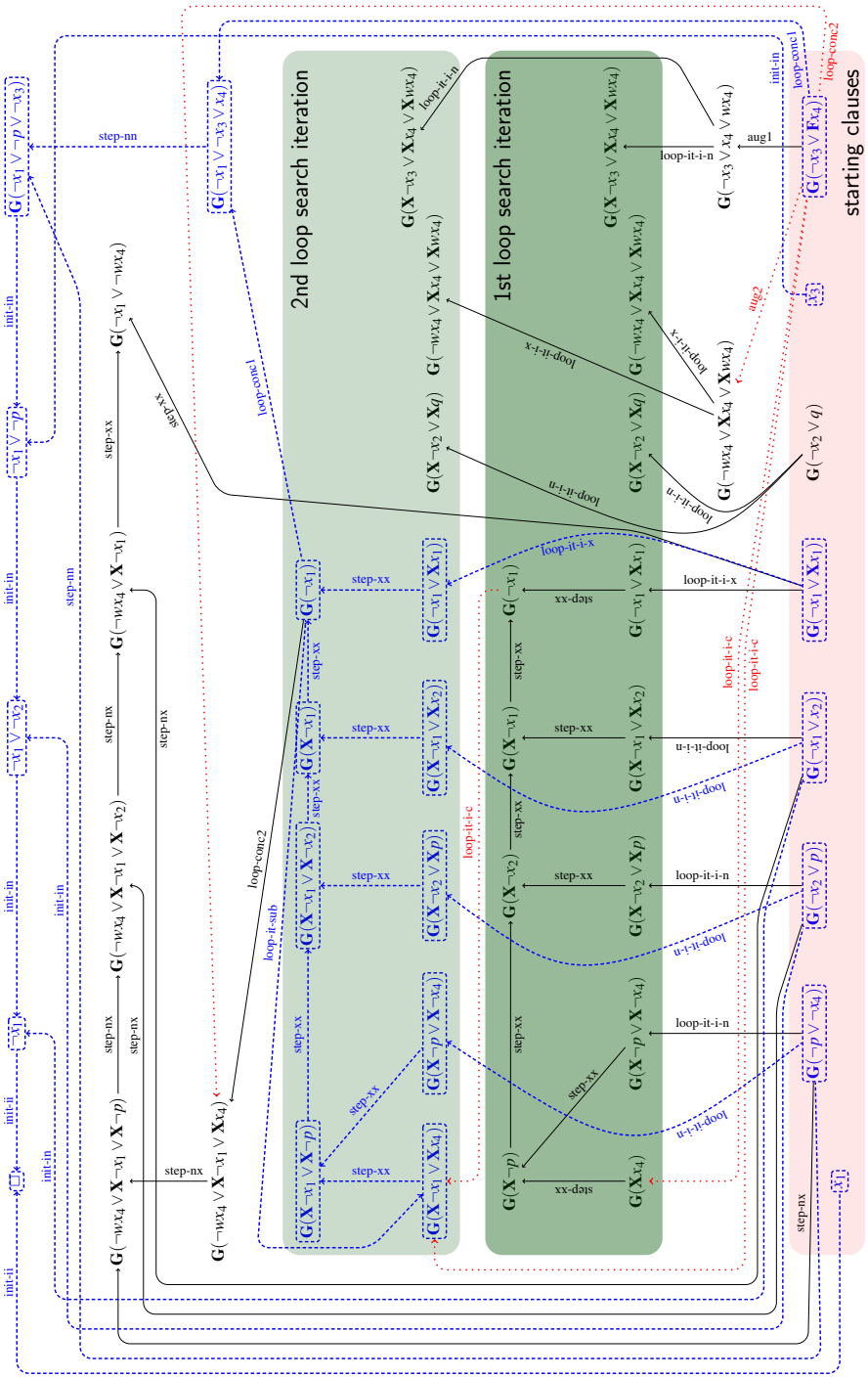



**Fig. 5** Example of an optimized resolution graph and UC extraction in SNF from an optimized resolution graph

blue boxed in Tab. 1. (We are sloppy in that we "replace" subformulas of replaced subformulas, while in effect they simply vanish.)

**Theorem 3 (Unsatisfiability of UC in LTL from SNF)** *Let $\phi$ be an unsatisfiable LTL formula, and let $\phi^{uc}$ be a UC of $\phi$ in LTL from SNF. Then $\phi^{uc}$ is unsatisfiable.*

*Proof* Let $SNF(\phi)$ be the SNF of $\phi$, and let $C^{uc}$ be a UC of $SNF(\phi)$ in SNF. Let $C^{uc'}$ be the UC of $C^{uc}$ in SNF via TR from an optimized resolution graph.

First, consider the trivial case that $\phi$ is FALSE. Here, Def. 10 results in the UC of $\phi$ in LTL being $\phi^{uc} \equiv$ FALSE as desired.

Now assume that $\phi$ is not FALSE, i.e., the size of the syntax tree of $\phi$ is greater than 1. Let $SNF(\phi^{uc})$ be the SNF of $\phi^{uc}$. In order to prove that $\phi^{uc}$ is unsatisfiable we show that the clauses of $C^{uc'}$ (which is unsatisfiable by Def. 5 and Thm. 2) are a subset of the SNF of $\phi^{uc}$: $C^{uc'} \subseteq SNF(\phi^{uc})$.

By comparing the clauses of $SNF(\phi)$ with those of $SNF(\phi^{uc})$ we can partition the clauses of $SNF(\phi)$ into three sets:[10] (i) Some clauses are present in both $SNF(\phi)$ and $SNF(\phi^{uc})$: $C'_1 \equiv SNF(\phi) \cap SNF(\phi^{uc})$. (ii) Some clauses are present in $SNF(\phi)$ and are present in $SNF(\phi^{uc})$ with one or more occurrences of some propositions $x, x', \ldots$ that are marked blue boxed in Tab. 1 replaced with TRUE or FALSE. Call that set $C'_2$. (iii) Some clauses are present in $SNF(\phi)$ but not in $SNF(\phi^{uc})$: $C'_3 \equiv SNF(\phi) \setminus (SNF(\phi^{uc}) \cup C'_2)$.

By Def. 5 and Cor. 2 $C^{uc'}$ is a subset of $SNF(\phi)$: $C^{uc'} \subseteq C^{uc} \subseteq SNF(\phi)$.

By Def. 10 $C^{uc}$ and, therefore, also $C^{uc'}$ contains no member of $C'_2$; otherwise, there could not be one or more occurrences of some propositions $x, x', \ldots$ that are marked blue boxed in Tab. 1 replaced with TRUE or FALSE in the clauses of $C'_2$: $C^{uc'} \cap C'_2 = \emptyset$.

Now we argue that $C^{uc'}$ also contains no member of $C'_3$. First, let $c \in C'_3$ be an initial or a global clause. $c$ cannot be a member of $C^{uc'}$ as, in order to be part of a proof that derives the empty clause, all literals of $c$ need to be "resolved away". However, this is not possible for $c$ as for the literal $(\neg)x_\psi$ on the left side of the implication in Tab. 1 there is no clause with an opposite literal in $C^{uc'}$. This follows by induction on the nesting depth of the subformula $\psi$ to which $(\neg)x_\psi$ belongs from the occurrence of the superformula of $\psi$ that has been replaced with TRUE or FALSE in $\phi^{uc}$. Now let $c \in C'_3$ be an eventuality clause. By Def. 8 for such $c$ to be part of $C^{uc'}$ there would have to be a clause $c'$ in the resolution graph $G$ according to Def. 2, 3, 7 that was generated by production rules aug1 or BFS-loop-conclusion1 and that is backward reachable in $G$ from the vertex labeled with the empty clause $\square$ in the main partition $M$, $v_\square$. Again, for the latter to happen, all literals of $c'$ would have to be "resolved away", which is impossible by a similar inductive argument as before.

Hence, we have shown that all clauses in $C^{uc'}$ come from $C'_1$, which is a subset of $SNF(\phi^{uc})$. This concludes the proof. $\square$

As a UC in LTL from SNF fulfills requirement (i) in Def. 9, we obtain Cor. 3.

**Corollary 3 (UC in LTL from SNF is UC in LTL)** *Let $\phi$ be an unsatisfiable LTL formula, and let $\phi^{uc}$ be a UC of $\phi$ in LTL from SNF. Then $\phi^{uc}$ is a UC of $\phi$ in LTL.*

*Remark 8 (Determinants of UC in LTL from SNF)* Note that an unsatisfiable LTL formula $\phi$ may have several UCs in LTL. If the UC of $\phi$ is obtained via Def. 10, 8, then Rem. 5 on the determinants of a UC in SNF applies here, too. Moreover, using a different translation from LTL into SNF instead of Def. 1 might also lead to a different UC in LTL from SNF.

[10] We disregard the issue of the indices of the variables $x, x', \ldots$.

We now complete the running example from Sec. 3 and the previous subsection. We show how to obtain a UC of $\phi$ in LTL from SNF from the UC of $\phi$ in SNF. Remember that the formula $\phi$, which we would like to obtain a UC of, is $(\mathbf{G}(p \wedge q)) \wedge \mathbf{F}\neg p$ (cf. (10)). Moreover, the SNF of $\phi$, $C$, on which in Fig. 5 we executed the algorithm in Fig. 3, is $\{x_1, \mathbf{G}(\neg x_1 \vee \mathbf{X}x_1), \mathbf{G}(\neg x_1 \vee x_2), \mathbf{G}(\neg x_2 \vee p), \mathbf{G}(\neg x_2 \vee q), x_3, \mathbf{G}(\neg x_3 \vee \mathbf{F}x_4), \mathbf{G}(\neg p \vee \neg x_4)\}$ (cf. (12)). Finally, the UC of $\phi$ in SNF, $C^{uc}$, is $C \setminus \{\mathbf{G}(\neg x_2 \vee q)\}$ (cf. (13)). Using the correspondence of $x_1$ to $x_{\mathbf{G}(p \wedge q)}$, of $x_2$ to $x_{p \wedge q}$, of $x_3$ to $x_{\mathbf{F}\neg p}$, and of $x_4$ to $x_{\neg p}$ as well as the definition of implication we rewrite[11] the UC of $\phi$ in SNF, $C$ to (19).

$$\begin{array}{l}\{(x_{\mathbf{G}(p \wedge q)}),\\ \ (\mathbf{G}(x_{\mathbf{G}(p \wedge q)} \rightarrow \mathbf{X}x_{\mathbf{G}(p \wedge q)})),\\ \ (\mathbf{G}(x_{\mathbf{G}(p \wedge q)} \rightarrow x_{p \wedge q})),\\ \ (\mathbf{G}(x_{p \wedge q} \rightarrow p)),\\ \ (x_{\mathbf{F}\neg p}),\\ \ (\mathbf{G}(x_{\mathbf{F}\neg p} \rightarrow \mathbf{F}x_{\neg p})),\\ \ (\mathbf{G}(x_{\neg p} \rightarrow \neg p))\}\end{array} \tag{19}$$

By careful inspection of (19) we see that $q$ is the only subformula of $\phi$ whose proposition according to column 2 of Tab. 1, which is $q$ itself, does not occur in any clause of (19) in a position that is marked blue boxed in Tab. 1. Hence, $q$ is the only subformula to be replaced by TRUE or FALSE in $\phi$, yielding the UC of $\phi$ in LTL from SNF, $\phi^{uc}$, in (20).

$$\phi^{uc} \equiv (\mathbf{G}(p \wedge \text{TRUE})) \wedge \mathbf{F}\neg p \tag{20}$$

## 5 Post-Processing UCs

In this section we adapt two techniques to our setting of UCs for LTL that can be used to make the UCs obtained so far more useful. Both techniques will typically be applied after an initial UC has been obtained; hence, we term this section post-processing of UCs. First, we discuss minimality of UCs. Then we show how to partition occurrences of propositions in a UC according to whether they interact in a TR proof of unsatisfiability, leading to a more fine-grained notion of UC. For a more advanced method of post-processing, which extracts information on the time points at which occurrences of subformulas are relevant to unsatisfiability, see [84].

### 5.1 Minimal UCs

In this subsection we introduce notions of and algorithms to obtain minimal UCs. The results are either straightforward (Prop. 4) or well known (Def. 11, Rem. 9). Still, the material is needed in the experimental evaluation, and within the flow of the article this seems to be the appropriate place.

[11] Notice that this is done purely for the convenience of the reader and does not correspond to a step in our method for UC extraction.

**Definition 11 (Minimal UC in SNF and LTL)** (See, e.g., [83]: irreducible UC) A UC $C^{uc}$ in SNF is *minimal* iff $\forall c \in C^{uc} \,.\, C^{uc} \setminus \{c\}$ is satisfiable. A UC $\phi^{uc}$ in LTL is *minimal* iff there is no positive polarity occurrence of a subformula that can be replaced with TRUE and no negative polarity occurrence of a subformula that can be replaced with FALSE without making $\phi^{uc}$ satisfiable.

**Proposition 4 (Minimal UC in SNF No Guarantee for Minimal UC in LTL)** *Let $\phi$ be an unsatisfiable LTL formula, let $SNF(\phi)$ be its SNF, let $C^{uc}$ be a minimal UC of $SNF(\phi)$ in SNF, and let $\phi^{uc}$ be the UC of $\phi$ in LTL from SNF. Then $\phi^{uc}$ is not necessarily minimal.*

*Proof* Let $\phi \equiv (\neg p) \wedge ((\mathbf{G}\neg q) \wedge (p\mathbf{U}q))$. Then

$$
\begin{aligned}
SNF(\phi) \equiv\ & \{(x_\phi),\\
& (\mathbf{G}(x_\phi \rightarrow x_{\neg p})),\\
& (\mathbf{G}(x_{\neg p} \rightarrow \neg p)),\\
& (\mathbf{G}(x_\phi \rightarrow x_{(\mathbf{G}\neg q)\wedge(p\mathbf{U}q)})),\\
& (\mathbf{G}(x_{(\mathbf{G}\neg q)\wedge(p\mathbf{U}q)} \rightarrow x_{\mathbf{G}\neg q})),\\
& (\mathbf{G}(x_{\mathbf{G}\neg q} \rightarrow \mathbf{X}x_{\mathbf{G}\neg q})),\\
& (\mathbf{G}(x_{\mathbf{G}\neg q} \rightarrow x_{\neg q})),\\
& (\mathbf{G}(x_{\neg q} \rightarrow \neg q)),\\
& (\mathbf{G}(x_{(\mathbf{G}\neg q)\wedge(p\mathbf{U}q)} \rightarrow x_{p\mathbf{U}q})),\\
& (\mathbf{G}(x_{p\mathbf{U}q} \rightarrow (q \vee p))),\\
& (\mathbf{G}(x_{p\mathbf{U}q} \rightarrow (q \vee \mathbf{X}x_{x_{p\mathbf{U}q}}))),\\
& (\mathbf{G}(x_{p\mathbf{U}q} \rightarrow \mathbf{F}q))\}
\end{aligned}
$$

is its SNF according to Def. 1. A minimal UC of $SNF(\phi)$ in SNF is

$$
\begin{aligned}
C^{uc} \equiv\ & \{(x_\phi),\\
& (\mathbf{G}(x_\phi \rightarrow x_{\neg p})),\\
& (\mathbf{G}(x_{\neg p} \rightarrow \neg p)),\\
& (\mathbf{G}(x_\phi \rightarrow x_{(\mathbf{G}\neg q)\wedge(p\mathbf{U}q)})),\\
& (\mathbf{G}(x_{(\mathbf{G}\neg q)\wedge(p\mathbf{U}q)} \rightarrow x_{\mathbf{G}\neg q})),\\
& (\mathbf{G}(x_{\mathbf{G}\neg q} \rightarrow x_{\neg q})),\\
& (\mathbf{G}(x_{\neg q} \rightarrow \neg q)),\\
& (\mathbf{G}(x_{(\mathbf{G}\neg q)\wedge(p\mathbf{U}q)} \rightarrow x_{p\mathbf{U}q})),\\
& (\mathbf{G}(x_{p\mathbf{U}q} \rightarrow (q \vee p)))\}.
\end{aligned}
$$

Mapping $C^{uc}$ back to a UC in LTL from SNF via Def. 10 yields $\phi$. $\phi$ is not a minimal UC, as the first conjunct $\neg p$ can be replaced with TRUE while retaining unsatisfiability. This concludes the proof. □

Note that given $\phi$ from the proof of Prop. 4 our implementation actually produces $\phi$ as a UC in LTL. This is due to the fact that the UC in SNF, $C^{uc}$, is found during the first execution of saturation in line 2 of the algorithm in Fig. 3, while the contradiction between $\mathbf{G}\neg q$ and the eventuality part of $p\mathbf{U}q$ requires loop search, which is only performed at a later stage.

Note also that the result just proved depends on the notion of UC for LTL: the proof above obviously does not hold, if the notion of UC allows to not only replace $\mathbf{G}\neg q$ with TRUE but also, alternatively, with $\neg q$ and $p\mathbf{U}q$ not only with TRUE but also, alternatively, with $p \vee q$.

*Remark 9 (Extraction of Minimal UCs)* A common way to obtain minimal UCs works by repeatedly attempting to remove parts of a UC (e.g., [20, 6, 99, 68, 3, 95]). If the modified formula is still unsatisfiable, then the removal is made permanent; otherwise the removal is undone. The procedure continues until all parts of the UC have been considered for removal. This is called *deletion-based extraction of minimal UCs* (e.g., [20, 68]).

In the case of LTL the algorithm attempts to replace positive polarity occurrences of subformulas with TRUE and negative polarity ones with FALSE. It terminates, if no more replacements can be performed without making the resulting formula satisfiable.

Naturally, this method may be expensive due to the number of satisfiability tests to be performed. It is therefore often used to minimize a UC that has been obtained by other means such as those described in Sec. 4 (see, e.g., [20, 6, 99, 68, 3, 95]). Potential optimizations of the minimization algorithm include binary search (e.g., [98, 58, 68, 61]) and reusing intermediate results (e.g., [95, 61]).

## 5.2 Grouping Propositions in UCs

In this subsection we show how to extract information from a TR proof of unsatisfiability on which occurrences of same propositions in a UC actually need to be the same propositions to retain unsatisfiability and which occurrences of same propositions might be substituted with different propositions without losing unsatisfiability. This can provide helpful information on the interaction of parts of a formula to a user who is debugging a specification.

### *5.2.1 Intuition*

As an example consider

$$(p \wedge \neg p) \vee \mathbf{X}((\mathbf{G}p) \wedge \mathbf{F}\neg p). \tag{21}$$

Note that (21) is a minimal UC in LTL. However, from the point of view of unsatisfiability of (21), the occurrences of $p$ in $p \wedge \neg p$ "have nothing to do" with the occurrences of $p$ in $\mathbf{X}((\mathbf{G}p) \wedge \mathbf{F}\neg p)$. On the other hand, the first occurrence of $p$ in $p \wedge \neg p$ must be an occurrence of the same proposition as the second occurrence of $p$ in $p \wedge \neg p$ to obtain unsatisfiability. The same is true for the two occurrences of $p$ in $\mathbf{X}((\mathbf{G}p) \wedge \mathbf{F}\neg p)$. Hence, the first two occurrences of $p$ in (21) could be substituted with, e.g., $p_0$ and the second two occurrences with, e.g., $p_1$, obtaining (22). (22) retains unsatisfiability and the structure of (21).

$$(p_0 \wedge \neg p_0) \vee \mathbf{X}((\mathbf{G}p_1) \wedge \mathbf{F}\neg p_1) \tag{22}$$

Note that, depending on the intended application of the UCs, (22) may or may not be considered to be a valid notion of UC of (21). For the purpose of debugging specifications we believe that this more fine-grained notion of UC can provide helpful additional information to the user.

We call (22) a UC of (21) in LTL *with grouped propositions*. We use the term "group" as a synonym to "partition" in order to avoid confusion with the notion of "partition" of vertices in the resolution graph in Sec. 4.1.

The information required to construct a UC with grouped propositions is obtained by observing the interaction of occurrences of propositions in a TR proof. Essentially, if two occurrences of the same proposition $p$ in an LTL formula $\phi$ are found not to be interacting in a TR proof of the unsatisfiability of $\phi$, then these occurrences of $p$ can be substituted with different propositions in a UC of $\phi$ without losing unsatisfiability.

Let $o$ and $o'$ be two occurrences of a proposition $p$ in two SNF clauses $c$ and $c'$. The occurrences $o$ and $o'$ in $c$ and $c'$ can interact in four different ways:

1. $c$ might be a premise and $c'$ might be a conclusion that is obtained from $c$ (and possibly some other clauses) by application of a production rule from Tab. 2. Then, some propositions may be transferred from $c$ to $c'$. Occurrences of propositions that are subject to transfer from $c$ to $c'$ are said to interact. For example, assume that $(p \vee q)$ and $(\mathbf{G}((\neg q) \vee r))$ are resolved to $(p \vee r)$ using $\boxed{\text{init-in}}$. Then the occurrence of $p$ in the premise $(p \vee q)$ is transferred to the occurrence of $p$ in the conclusion $(p \vee r)$, as is the occurrence of $r$ in the premise $(\mathbf{G}((\neg q) \vee r))$ to the occurrence of $r$ in the conclusion $(p \vee r)$.
2. $c$ and $c'$ might be resolved with each other using one of the saturation rules in Tab. 2. Then the two occurrences of the proposition which is resolved upon (which have different polarity) are also said to interact. In the example above the occurrences of $q$ in $(p \vee q)$ and in $(\mathbf{G}((\neg q) \vee r))$ interact.
3. $c$ and $c'$ might be eventuality clauses with the same eventuality literal. Then the two occurrences of the eventuality literal are said to interact.
4. There might be a finite sequence of interactions of the former three kinds linking the occurrences $o$ and $o'$ in $c$ and $c'$, i.e., we form the transitive closure of the interaction relation.

### *5.2.2 Formalization — SNF*

In Def. 12–14 we formalize the idea as follows.[12] Each occurrence of a proposition in the resolution graph is mapped to some group (here groups are arbitrarily represented by natural numbers). If two occurrences of a proposition interact, then they are forced to be mapped to the same group. In our example the occurrences of $p$ in $(p \vee q)$ and in $(p \vee r)$ might be mapped to $i \in \mathbb{N}$, the occurrences of $q$ in $(p \vee q)$ and in $(\mathbf{G}((\neg q) \vee r))$ to $i' \in \mathbb{N}$ different from $i$, and the occurrences of $r$ in $(\mathbf{G}((\neg q) \vee r))$ and in $(p \vee r)$ to $i'' \in \mathbb{N}$ different from both $i$ and $i'$. This leads to a partitioning of the occurrences of propositions in clauses labeling vertices in the resolution graph. Occurrences in each partition are then replaced with a different proposition.

**Definition 12 (Grouping of Propositions in Resolution Graph)** Let $C \subseteq \mathbb{C}$ be a set of SNF clauses, and let $G$ be a resolution graph with set of vertices $V$ and labeling of vertices with SNF clauses $L_V$. Let $\mathscr{O}$ be the set of occurrences of propositions in clauses in $\mathbb{C}$. Let *group* be a mapping from pairs of vertices in $V$ and occurrences of propositions in clauses in $\mathbb{C}$ to groups in $\mathbb{N}$, $group : V \times \mathscr{O} \to \mathbb{N}$, fulfilling the following four conditions.

1. If (i) $v_0$ is a vertex in $V$ labeled with a clause $c_0$: $v_0 \in V$ and $L_V(v_0) = c_0$, (ii) $v_1$ is a vertex in $V$ labeled with a clause $c_1$: $v_1 \in V$ and $L_V(v_1) = c_1$, (iii) $c_1$ is a conclusion obtained from premise $c_0$ (and possibly other premises) by an application of a production rule $\boxed{\text{rule}}$ from Tab. 2 in $G$, (iv) $o_0$ is an occurrence of a proposition $p$ in $c_0$, (v) $o_1$ is an occurrence of a proposition $q$ in $c_1$, and (vi) $\boxed{\text{rule}}$ constrains $p$ in $o_0$ and $q$ in $o_1$ to be the same proposition, then $(v_0, o_0)$ and $(v_1, o_1)$ are mapped to the same group: $group((v_0, o_0)) = group((v_1, o_1))$.

[12] Remember that in this section we only consider optimized resolution graphs and UCs in SNF via TR from optimized resolution graphs, and we drop the designators "optimized" and "from an optimized resolution graph". Moreover, we drop general definitions and statements in the style of Def. 5 and Cor. 2 and restrict ourselves to the specific cases á la Def. 8 and Thm. 2.

2. If (i) $v_2$ is a vertex in $V$ labeled with a clause $c_2$: $v_2 \in V$ and $L_V(v_2) = c_2$, (ii) $v_3$ is a vertex in $V$ labeled with a clause $c_3$: $v_3 \in V$ and $L_V(v_3) = c_3$, (iii) $c_2$ and $c_3$ are premises in the application of a production rule $\boxed{\text{rule}} \in \{\boxed{\text{init-ii}}, \boxed{\text{init-in}}, \boxed{\text{step-nn}}, \boxed{\text{step-nx}}, \boxed{\text{step-xx}}\}$ from Tab. 2 in $G$, (iv) $o_2$ is the occurrence of the proposition $r$ in $c_2$ that is resolved upon in the application of $\boxed{\text{rule}}$, and (v) $o_3$ is the occurrence of the proposition $r$ in $c_3$ that is resolved upon in the application of $\boxed{\text{rule}}$, then $(v_2, o_2)$ and $(v_3, o_3)$ are mapped to the same group: $group((v_2, o_2)) = group((v_3, o_3))$.
3. If (i) $v_4$ is a vertex in $V$ labeled with an eventuality clause $c_4$: $v_4 \in V$ and $L_V(v_4) = c_4$, (ii) $v_5$ is a vertex in $V$ labeled with an eventuality clause $c_5$: $v_5 \in V$ and $L_V(v_5) = c_5$, (iii) $o_4$ is the occurrence of the eventuality literal $l$ in $c_4$, (iv) $o_5$ is the occurrence of the eventuality literal $l$ in $c_5$, then $(v_4, o_4)$ and $(v_5, o_5)$ are mapped to the same group: $group((v_4, o_4)) = group((v_5, o_5))$.
4. If two pairs $(v_6, o_6)$ and $(v_7, o_7)$ are not (transitively) forced to be mapped to the same group by the former three conditions, then they are mapped to different groups.

Then *group* is a *grouping of propositions in a resolution graph*.

**Definition 13 (Grouping-Induced Substitution of Propositions in Resolution Graph)** Let $C \subseteq \mathbb{C}$ be a set of SNF clauses, and let $G$ be a resolution graph with set of vertices $V$ and labeling of vertices with SNF clauses $L_V$. Let $\mathscr{O}$ be the set of occurrences of propositions in clauses in $\mathbb{C}$. Let *group* be the grouping of propositions in $G$. Let $f$ be an injective mapping from the image of *group* to propositions $AP$: $f : \{i \in \mathbb{N} \mid \exists v \in V \,.\, \exists o \text{ in } L_V(v) \,.\, i = group((v, o))\} \to AP$ such that $f(i) = f(i') \Rightarrow i = i'$. Then the composition of $f$ and *group*, $subst = f \circ group$, is a *grouping-induced substitution of propositions in a resolution graph*.

We extend the domain of *subst* to clauses in $G$ and to $G$ itself in the natural way. If the vertex $v_c$ that a clause $c$ is labeling is clear from the context, we write $subst(c)$ instead of $subst(v_c, c)$.

**Definition 14 (UC in SNF via TR with Grouped Propositions)** Let $C$ be an unsatisfiable set of SNF clauses, and let $C^{uc}$ be a UC of $C$ in SNF via TR. Let *subst* be the grouping-induced substitution of propositions in a resolution graph. The *UC of $C$ in SNF via TR with grouped propositions*, $C^{uc}_{group}$, is obtained from the UC in SNF via TR $C^{uc}$ by applying *subst* to each clause in $C^{uc}$: $C^{uc}_{group} = \{subst(c) \mid c \in C^{uc}\}$.

Assume that Def. 12–14 are applied to

$$C \equiv \{(p \vee q), (\neg r), (\mathbf{G}((\neg p) \vee \mathbf{X} r)), (\mathbf{G}\,\mathbf{X} \neg r), (\mathbf{G}((\neg q) \vee r))\}. \tag{23}$$

Now it is easy to see that there exists a TR proof of the unsatisfiability of $C$ such that both occurrences of $p$ are mapped to some group $i$, both occurrences of $q$ are mapped to a different group $i'$, the occurrences of $r$ in $(\neg r)$ and in $(\mathbf{G}((\neg q) \vee r))$ are mapped to another group $i''$, and, finally, the remaining occurrences of $r$ in $(\mathbf{G}((\neg p) \vee \mathbf{X} r))$ and in $(\mathbf{G}\,\mathbf{X} \neg r)$ are mapped to a last group $i'''$. To obtain a UC of $C$ in SNF via TR with grouped propositions we substitute occurrences of propositions according to the groups they are mapped to: occurrences of propositions are substituted with the same proposition if and only if they are mapped to the same group. In the example (23) we obtain the following UC (24):

$$C^{uc}_{group} \equiv \{(p \vee q), (\neg r_0), (\mathbf{G}((\neg p) \vee \mathbf{X} r_1)), (\mathbf{G}\,\mathbf{X} \neg r_1), (\mathbf{G}((\neg q) \vee r_0))\}. \tag{24}$$

The proof of correctness in Thm. 4 is essentially by saying that the TR proof for the unsatisfiability of a UC in SNF via TR is also a TR proof for the unsatisfiability of a UC in

SNF via TR with grouped propositions modulo renaming of some occurrences of propositions. Before stating Thm. 4 we mention some properties of *group* and *subst* required in its proof of correctness.

*Remark 10 (group Maps Different Propositions to Different Groups)* Let $C$ be a set of SNF clauses, let $G$ be a resolution graph with set of vertices $V$ and labeling of vertices with SNF clauses $L_V$, and let *group* be the grouping of propositions in a resolution graph. *group* maps occurrences of different propositions to different groups: if (i) $v$ is a vertex in $V$ labeled with a clause $c$: $v \in V$ and $L_V(v) = c$, (ii) $v'$ is a vertex in $V$ labeled with a clause $c'$: $v' \in V$ and $L_V(v') = c'$, (iii) $o$ is an occurrence of a proposition $p$ in $c$, (iv) $o'$ is an occurrence of a proposition $q$ in $c'$, and (v) $p \neq q$, then $group((v,o)) \neq group((v',o'))$. Given the injectivity of $f$ in Def. 13 this directly extends to *subst*.

*Remark 11 (subst Does Not Change Polarity of Literal)* Let $C$ be a set of SNF clauses, let $G$ be a resolution graph, and let *subst* be the grouping-induced substitution of propositions in a resolution graph. Then *subst* does not change the polarity of a literal in a clause.

*Remark 12 (subst Does Not Change Time of Literal)* Let $C$ be a set of SNF clauses, let $G$ be a resolution graph, and let *subst* be the grouping-induced substitution of propositions in a resolution graph. Then *subst* maps an initial literal to an initial literal, a literal in the now part to a literal in the now part, a literal in the **X** part to a literal in the **X** part, and an eventuality literal to an eventuality literal.

**Theorem 4 (Unsatisfiability of UC in SNF via TR with Grouped Propositions)** *Let $C$ be an unsatisfiable set of SNF clauses, and let $C^{uc}_{group}$ be a UC of $C$ in SNF via TR with grouped propositions. Then $C^{uc}_{group}$ is unsatisfiable.*

*Proof* Let $C^{uc}$ be the UC in SNF via TR. Let $G$ be the resolution graph with set of vertices $V$ and labeling of vertices with SNF clauses $L_V$. Let *subst* be the grouping-induced substitution of propositions in a resolution graph. Let $G_{group} = subst(G)$.

Possibly contrary to Def. 12, 13 we assume that *subst* maps occurrences of same propositions $wl_0$ introduced by augmentation aug1, aug2 to same propositions. Note that propositions $wl_0$ do not occur in $C^{uc}$ and, therefore, their images under *subst* do not occur in $C^{uc}_{group}$. Hence, if we can show unsatisfiability of $C^{uc}_{group}$ under this assumption, then we have shown unsatisfiability of $C^{uc}_{group}$.

Below we show that each application of a production rule rule to clauses labeling some vertices in $G$ is also an application of rule to the clauses labeling these vertices in $G_{group}$. This establishes that $G_{group}$ represents a (possibly partial) TR proof. Consider the following cases:

init-ii Let $v, v', v'' \in V$ with $L_V(v) = c$, $L_V(v') = c'$, and $L_V(v'') = c''$ where $c''$ is obtained from $c$ and $c'$ by applying init-ii. By Rem. 12 $subst(c)$, $subst(c')$, and $subst(c'')$ are initial clauses. Let $l_0$, $\neg l_0$ be the resolved upon literals in $c$ and $c'$. By Def. 12, 13 *subst* maps these occurrences of $l_0$ and $\neg l_0$ to opposite polarity occurrences of some proposition $p$ in $subst(c)$ and $subst(c')$. Let there be an occurrence of a not resolved upon literal $l_1 \neq p, \neg p$ in $subst(c)$ (resp. $subst(c')$). By Def. 12, 13 there is an occurrence $o$ of a literal $l_2$ in $c$ (resp. $c'$) such that $subst(v,o) = l_1$. By rule init-ii there is a corresponding occurrence of $l_2$ in $c''$. By Def. 12, 13 *subst* maps that occurrence of $l_2$ to an occurrence of $l_1$ in $subst(c'')$. Let there be an occurrence of a literal $l_3$ in $subst(c'')$. By Def. 12, 13 there is an occurrence $o'$ of a literal $l_4$ in $c''$ such that $subst(v'',o') = l_3$. By rule init-ii there is a corresponding occurrence of $l_4$ in $c$ or $c'$. By Def. 12, 13 *subst* maps that

occurrence of $l_4$ to an occurrence of $l_3$ in $subst(c)$ or $subst(c')$. This shows that $subst(c'')$ can be obtained by applying init-ii to $subst(c)$ and $subst(c')$.

init-in, step-nn Similar to the case of init-ii.

step-nx, step-xx Similar to the case of init-ii when considering now and **X** parts separately as appropriate.

aug1 Let $v, v' \in V$ with $L_V(v) = c$ and $L_V(v') = c'$ where $c'$ is obtained from $c$ by applying aug1, $l_0$ is the eventuality literal in $c$, and $wl_0$ is the fresh literal introduced by aug1 and aug2 for all eventuality clauses in $G$ with eventuality literal $l_0$. By Rem. 12 $subst(c)$ is an eventuality clause and $subst(c')$ is a global clause with empty **X** part. By Def. 12, 13 *subst* maps the occurrences of $l_0$ as the eventuality literal in $c$ and in $c'$ to two occurrences of some literal $l_1$ as the eventuality literal in $subst(c)$ and in $subst(c')$. By Def. 12, 13 *subst* maps the occurrence of $wl_0$ in $c'$ to an occurrence of some literal $wl_1$ in $subst(c')$. By Rem. 10 $wl_1$ is fresh. By Def. 12, 13 and the assumption above $wl_1$ is the fresh literal introduced by aug1 and aug2 for all eventuality clauses in $G_{group}$ with eventuality literal $l_1$. Let there be an occurrence of a literal $l_2$ in the now part of $subst(c)$. By Def. 12, 13 there is an occurrence $o$ of a literal $l_3$ in the now part of $c$ such that $subst(v, o) = l_2$. By rule aug1 there is a corresponding occurrence of $l_3$ in $c'$. By Def. 12, 13 *subst* maps that occurrence of $l_3$ to an occurrence of $l_2$ in $subst(c')$. Let there be an occurrence of a literal $l_4 \neq l_1, wl_1$ in $subst(c')$. By Def. 12, 13 there is an occurrence $o'$ of a literal $l_5$ in $c'$ such that $subst(v', o') = l_4$. By rule aug1 there is a corresponding occurrence of $l_5$ in the now part of $c$. By Def. 12, 13 *subst* maps that occurrence of $l_5$ to an occurrence of $l_4$ in the now part of $subst(c)$. This shows that $subst(c')$ can be obtained by applying aug1 to $subst(c)$.

aug2 Let $v \in V$ with $L_V(v) = c$ where $c$ is obtained by applying aug2 to eventuality clauses with eventuality literal $l_0$. By Def. 12, 13 and the assumption above $subst(c)$ is of the form $(\mathbf{G}((\neg wl_1) \vee \mathbf{X}(l_1 \vee wl_1)))$ for some $l_1$, $wl_1$. By Def. 12, 13 $l_1$ is unique for all occurrences of $l_0$ as eventuality literal in $G$. By Rem. 10 $wl_1$ is fresh. By Def. 12, 13 and the assumption above $wl_1$ is the fresh literal introduced by aug1 and aug2 for all eventuality clauses in $G_{group}$ with eventuality literal $l_1$. This shows that $subst(c)$ can be obtained by applying aug2 to eventuality clauses with eventuality literal $l_1$.

BFS-loop-it-init-x Let $v, v' \in V$ with $L_V(v) = c$ and $L_V(v') = c'$ where $c'$ is obtained from $c$ by applying BFS-loop-it-init-x. By Rem. 12 $subst(c)$ and $subst(c')$ are global clauses with non-empty **X** part. Let there be an occurrence of a literal $l_0$ in the now (resp. **X**) part of $subst(c)$. By Def. 12, 13 there is an occurrence $o$ of a literal $l_1$ in the now (resp. **X**) part of $c$ such that $subst(v, o) = l_0$. By rule BFS-loop-it-init-x there is a corresponding occurrence of $l_1$ in the now (resp. **X**) part of $c'$. By Def. 12, 13 *subst* maps that occurrence of $l_1$ to an occurrence of $l_0$ in the now (resp. **X**) part of $subst(c')$. Let there be an occurrence of a literal $l_2$ in the now (resp. **X**) part of $subst(c')$. By Def. 12, 13 there is an occurrence $o'$ of a literal $l_3$ in the now (resp. **X**) part of $c'$ such that $subst(v', o') = l_2$. By rule BFS-loop-it-init-x there is a corresponding occurrence of $l_3$ in the now (resp. **X**) part of $c$. By Def. 12, 13 *subst* maps that occurrence of $l_3$ to an occurrence of $l_2$ in the now (resp. **X**) part of $subst(c)$. This shows that $subst(c')$ can be obtained by applying BFS-loop-it-init-x to $subst(c)$.

BFS-loop-it-init-n Similar to the case of BFS-loop-it-init-x.

BFS-loop-it-init-c First, consider the case that the current BFS loop search iteration is the second or later iteration of a BFS loop search. Let $v, v', v'' \in V$ with $L_V(v) = c$, $L_V(v') = c'$, and $L_V(v'') = c''$ where $c''$ is obtained from global clause with empty **X** part $c$ and eventuality clause $c'$ with eventuality literal $l_0$ by applying BFS-loop-it-init-c. By Rem. 12 $subst(c)$ is a global clause with empty **X** part, $subst(c')$ is an eventuality clause, and

$subst(c'')$ is a global clause with empty now part. By rule BFS-loop-it-init-c there is a corresponding occurrence of $l_0$ in $c''$. By Def. 12, 13 *subst* maps these occurrences of $l_0$ to occurrences of some literal $l_1$ as the eventuality literal in $subst(c')$ and as a disjunct in $subst(c'')$. Let there be an occurrence of a literal $l_2$ in $subst(c)$. By Def. 12, 13 there is an occurrence $o$ of a literal $l_3$ in $c$ such that $subst(v,o) = l_2$. By rule BFS-loop-it-init-c there is a corresponding occurrence of $l_3$ in $c''$. By Def. 12, 13 *subst* maps that occurrence of $l_3$ to an occurrence of $l_2$ in $subst(c'')$. Let there be an occurrence of a literal $l_4 \neq l_1$ in $subst(c'')$. By Def. 12, 13 there is an occurrence $o'$ of a literal $l_5$ in $c''$ such that $subst(v'',o') = l_4$. By rule BFS-loop-it-init-c there is a corresponding occurrence of $l_5$ in $c$. By Def. 12, 13 *subst* maps that occurrence of $l_5$ to an occurrence of $l_4$ in $subst(c)$. Note that following the reasoning in Thm. 2 it is not necessary to show that $subst(c)$ originated in the previous BFS loop search iteration. Hence, this shows that $subst(c'')$ can be obtained by applying BFS-loop-it-init-c to $subst(c)$ and $subst(c')$.

Now consider the case that the current BFS loop search iteration is the first iteration of a BFS loop search. In that case there essentially is no premise 1 $c$. Hence, this case is a trivial special case of the previous case.

BFS-loop-it-sub Let $v,v' \in V$ with $L_V(v) = c$ and $L_V(v') = c'$ where $c'$ is "obtained" from $c$ by applying BFS-loop-it-sub. Let $o$ be the occurrence of the eventuality literal $l_0$ in $c'$ that the loop search in $G$ is for. Let $subst(v',o) = l_1$. By Rem. 12 $subst(c)$ is a global clause with empty $\mathbf{X}$ part and $subst(c')$ is a global clause with empty now part. Using an inductive argument on the sequence in which clauses labeling vertices in $G$ are generated by the algorithm in Fig. 3 one can see that $subst(c')$ was generated by an application of BFS-loop-it-init-c. Let there be an occurrence of a literal $l_2$ in $subst(c)$. By Def. 12, 13 there is an occurrence $o'$ of a literal $l_3$ in $c$ such that $subst(v,o') = l_2$. By rule BFS-loop-it-sub there is a corresponding occurrence of $l_3 \neq l_0$ in $c'$. By Def. 12, 13 *subst* maps that occurrence of $l_3$ to an occurrence of $l_2 \neq l_1$ in $subst(c')$. This shows that $subst(c')$ can be obtained by applying BFS-loop-it-sub to $subst(c)$.

BFS-loop-conclusion1 Let $v,v',v'' \in V$ with $L_V(v) = c$, $L_V(v') = c'$, and $L_V(v'') = c''$ where $c''$ is obtained by applying BFS-loop-conclusion1 to a global clause with empty $\mathbf{X}$ part $c$ derived in a successful BFS loop search iteration and an eventuality clause $c'$ with eventuality literal $l_0$. By Rem. 12 $subst(c')$ is an eventuality clause and $subst(c)$ and $subst(c'')$ are global clauses with empty $\mathbf{X}$ part. By rule BFS-loop-conclusion1 there is a corresponding occurrence of $l_0$ in $c''$. By Def. 12, 13 *subst* maps these occurrences of $l_0$ to occurrences of some literal $l_1$ as the eventuality literal in $subst(c')$ and as a disjunct in $subst(c'')$. Let there be an occurrence of a literal $l_2$ in the now part of $subst(c)$ (resp. $subst(c')$). By Def. 12, 13 there is an occurrence $o$ of a literal $l_3$ in the now part of $c$ (resp. $c'$) such that $subst(v,o) = l_2$ (resp. $subst(v',o) = l_2$). By rule BFS-loop-conclusion1 there is a corresponding occurrence of $l_3$ in $c''$. By Def. 12, 13 *subst* maps that occurrence of $l_3$ to an occurrence of $l_2$ in $subst(c'')$. Let there be an occurrence of a literal $l_4 \neq l_1$ in $subst(c'')$. By Def. 12, 13 there is an occurrence $o'$ of a literal $l_5$ in $c''$ such that $subst(v'',o') = l_4$. By rule BFS-loop-conclusion1 there is a corresponding occurrence of $l_5$ in the now part of $c$ or $c'$. By Def. 12, 13 *subst* maps that occurrence of $l_5$ to an occurrence of $l_4$ in the now part of $subst(c)$ or $subst(c')$. Except for the requirement that $subst(c)$ was derived in a successful BFS loop search iteration this shows that $subst(c'')$ can be obtained by applying BFS-loop-conclusion1 to $subst(c)$ and $subst(c')$.

BFS-loop-conclusion2 Let $v,v',v'' \in V$ with $L_V(v) = c$, $L_V(v') = c'$, and $L_V(v'') = c''$ where $c''$ is obtained by applying BFS-loop-conclusion2 to a global clause with empty $\mathbf{X}$ part $c$ derived in a successful BFS loop search iteration and an eventuality clause $c'$ with eventuality literal $l_0$. By Rem. 12 $subst(c)$ is a global clause with empty $\mathbf{X}$ part, $subst(c')$ is an

eventuality clause, and $subst(c'')$ is a global clause. By rule BFS-loop-conclusion2 there is a corresponding occurrence of $l_0$ in the **X** part of $c''$. By Def. 12, 13 *subst* maps these occurrences of $l_0$ to occurrences of some literal $l_1$ as the eventuality literal in $subst(c')$ and as a disjunct in the **X** part of $subst(c'')$. By Def. 12, 13 *subst* maps the occurrence of $\neg wl_0$ in the now part of $c''$ to an occurrence of some literal $\neg wl_1$ in the now part of $subst(c'')$. By Def. 12, 13 and the assumption above $wl_1$ is the fresh literal introduced by aug1 and aug2 for all eventuality clauses in $G_{group}$ with eventuality literal $l_1$. Let there be an occurrence of a literal $l_2$ in $subst(c)$. By Def. 12, 13 there is an occurrence $o$ of a literal $l_3$ in $c$ such that $subst(v,o) = l_2$. By rule BFS-loop-conclusion2 there is a corresponding occurrence of $l_3$ in the **X** part of $c''$. By Def. 12, 13 *subst* maps that occurrence of $l_3$ to an occurrence of $l_2$ in the **X** part of $subst(c'')$. Let there be an occurrence of a literal $l_4 \neq l_1$ in the **X** part of $subst(c'')$. By Def. 12, 13 there is an occurrence $o'$ of a literal $l_5$ in the **X** part of $c''$ such that $subst(v'',o') = l_4$. By rule BFS-loop-conclusion2 there is a corresponding occurrence of $l_5$ in $c$. By Def. 12, 13 *subst* maps that occurrence of $l_5$ to an occurrence of $l_4$ in $subst(c)$. Except for the requirement that $subst(c)$ was derived in a successful BFS loop search iteration this shows that $subst(c'')$ can be obtained by applying BFS-loop-conclusion2 to $subst(c)$ and $subst(c')$.

It remains to show that a successful BFS loop search iteration in $G$ is also a successful BFS loop search iteration in $G_{group}$. Let $v \in V$ with $L_V(v) = c$ such that $subst(c)$ was generated by BFS-loop-it-init-c. As shown above $c$ was also generated by BFS-loop-it-init-c. If the BFS loop search iteration that $c$ is part of was successful, then there exists $v' \in V$ with $L_V(v') = c'$ such that $c'$ is part of the same BFS loop search iteration as $c$ and $c$ and $c'$ are connected by an application of BFS-loop-it-sub. As shown above $subst(c)$ and $subst(c')$ are connected by an application of BFS-loop-it-sub.

Clearly, for any clause $c$, $subst(c)$ cannot be larger than $c$. Moreover, with Rem. 10, $subst(c)$ cannot be smaller than $c$. Hence, the empty clause (and, therefore, unsatisfiability) is derived in the proof on $C^{uc}_{group}$ at the same place as in the proof on $C^{uc}$. This concludes the proof. □

### *5.2.3 Formalization — LTL*

In order to map a UC in SNF via TR with grouped propositions to a UC in LTL via TR with grouped propositions we need to take the translation from LTL into SNF and back in Sec. 3.3, 4.2 into account. Assume an occurrence of a proposition $p$ in a subformula $\psi$. Depending on $\psi$ that single occurrence of $p$ in $\psi$ may be translated into multiple occurrences of $p$ in $SNF(\psi)$. For example, if $\psi \equiv q\mathbf{U}p$ under positive polarity, then $\psi$ is translated into $(\mathbf{G}(x_\psi \rightarrow (p \vee q)))$, $(\mathbf{G}(x_\psi \rightarrow (p \vee \mathbf{X}x_\psi)))$, and $(\mathbf{G}(x_\psi \rightarrow \mathbf{F}p))$. Now, even if these occurrences of $p$ in the UC of $SNF(\psi)$ in SNF via TR with grouped propositions are mapped to different groups (and therefore are substituted with different propositions in the UC in SNF via TR with grouped propositions), we must ensure that the occurrence of $p$ in the UC in LTL via TR with grouped propositions is substituted with a unique proposition such that unsatisfiability of the result is guaranteed. This can be achieved by merging some groups in the mapping *group*: if two occurrences of a proposition $p$ in the set of starting clauses $SNF(\phi)$ were obtained from the same occurrence of $p$ in the LTL formula $\phi$, then these occurrences of $p$ must be mapped to the same group. Notice that now all occurrences of a proposition $p$ in the UC of $\phi$ in SNF via TR that were obtained from the same occurrence of $p$ in $\phi$ are mapped to the same group. This allows to transfer the grouping of occurrences of propositions from SNF to LTL. This is stated more formally in Def. 15–17.

**Definition 15 (Grouping of Propositions in Resolution Graph for LTL)** Let $\phi$ be an LTL formula, let $SNF(\phi)$ be the SNF of $\phi$, let $G$ be a resolution graph with set of vertices $V$ and labeling of vertices with SNF clauses $L_V$, and let $group$ be the grouping of propositions in a resolution graph. Let $group_{LTL}$ be obtained from $group$ by merging some groups as follows. Let $c$ and $c'$ be two clauses obtained from translating the occurrence of a subformula $\psi$ in $\phi$, let $v$ and $v'$ be the unique vertices in the main partition of $G$ labeled with $c$ and $c'$, and let $o$ and $o'$ be two occurrences of the same proposition in $c$ and $c'$ that are marked blue boxed in Tab. 1. Let $group((v,o)) = i$ and $group((v',o')) = i'$. Then all pairs $(v'',o'')$ in the domain of $group$ that are mapped to $i$ or to $i'$ in $group$ are mapped to the same group $i''$ in $group_{LTL}$. $group_{LTL}$ is the *grouping of propositions in a resolution graph for LTL*.

Inspection of Tab. 1 shows that the only propositions whose groups might be merged because of Def. 15 are propositions representing the right hand operand of a positive polarity occurrence of a **U** formula or of a negative polarity occurrence of a **R** formula.

**Definition 16 (Grouping-Induced Substitution of Propositions in Resolution Graph for LTL)** Let $\phi$ be an unsatisfiable LTL formula, let $SNF(\phi)$ be the SNF of $\phi$, let $C^{uc}$ be a UC of $SNF(\phi)$ in SNF via TR, let $G$ be the resolution graph with set of vertices $V$ and labeling of vertices with SNF clauses $L_V$, and let $group_{LTL}$ be the grouping of propositions in a resolution graph for LTL. Let $g$ be a mapping from occurrences of propositions in $\phi$ to sets of occurrences of propositions in $SNF(\phi)$ that maps an occurrence of a proposition $p$ in $\phi$ to the set of occurrences of $p$ in $SNF(\phi)$ that are induced by the occurrence of $p$ in $\phi$. Let $f_{LTL}$ be an injective mapping from the image of $group_{LTL}$ to propositions $AP$: $f_{LTL} : \{i \in \mathbb{N} \mid \exists v \in V \,.\, \exists o \text{ in } L_V(v) \,.\, i = group_{LTL}((v,o))\} \to AP$ such that $f_{LTL}(i) = f_{LTL}(i') \Rightarrow i = i'$. Extend $subst$ to LTL as follows. Use $g$ to map an occurrence $o$ of a proposition in $\phi^{uc}$ to a set of occurrences $O$ in $SNF(\phi)$. Choose an occurrence $o' \in O$ that is also an occurrence in $C^{uc}$. Use $f_{LTL} \circ group_{LTL}$ to map $o'$ to a proposition $p \in AP$. The resulting mapping $subst_{LTL}$ is the *grouping-induced substitution of propositions for LTL*.

We extend the domain of $subst_{LTL}$ from occurrences of propositions in $\phi^{uc}$ to $\phi^{uc}$ in the natural way.

**Definition 17 (UC in LTL via TR with Grouped Propositions)** Let $\phi$ be an unsatisfiable LTL formula, let $\phi^{uc}$ be its UC in LTL obtained using Def. 10, 8, and let $subst_{LTL}$ be the grouping-induced substitution of propositions in a resolution graph for LTL. Then $\phi^{uc}_{group} = subst_{LTL}(\phi^{uc})$ is the *UC of $\phi$ in LTL via TR with grouped propositions*.

**Theorem 5 (Unsatisfiability of UC in LTL via TR with Grouped Propositions)** *Let $\phi$ be an unsatisfiable LTL formula, and let $\phi^{uc}_{group}$ be a UC of $\phi$ in LTL via TR with grouped propositions. Then $\phi^{uc}_{group}$ is unsatisfiable.*

*Proof* (Sketch.) Let $\phi^{uc}$ be the UC of $\phi$ in LTL obtained using Def. 10, 8, let $SNF(\phi^{uc})$ be the SNF of $\phi^{uc}$, let $C^{uc}$ be the UC of $\phi$ in SNF via TR, let $SNF(\phi^{uc}_{group})$ be the SNF of $\phi^{uc}_{group}$, let $C^{uc}_{group}$ be the UC of $\phi$ in SNF via TR with grouped propositions, and let $C^{uc}_{group_{LTL}}$ be as $C^{uc}_{group}$ but using $group_{LTL}$ instead of $group$ in its construction. By the proof of Thm. 3 $SNF(\phi^{uc})$ is a superset of $C^{uc}$. By that fact and by the construction of $\phi^{uc}_{group}$ $SNF(\phi^{uc}_{group})$ is a superset of $C^{uc}_{group_{LTL}}$. By Thm. 4 $C^{uc}_{group}$ and, thus, $C^{uc}_{group_{LTL}}$ is unsatisfiable. Hence, so is $\phi^{uc}_{group}$. This concludes the proof. □

## 6 Relation to Mutual Vacuity

In this section we explain that, under the frequently legitimate assumption that a system description can be translated into an LTL formula, our results extend to vacuity for LTL [44, 9, 65, 2, 92, 40, 64]. In [44] Gurfinkel and Chechik introduce the notion of mutual vacuity. We prove that the problems of finding a UC of an unsatisfiable formula $\phi$ in LTL and of finding a set of subformula occurrences $O$ of an LTL specification $\phi$ such that $\phi$ is mutually vacuously TRUE in $O$ in a system $\zeta$ can be reduced to each other. For some more discussion on the relation between UCs and vacuity see also [83].

Given a system description $\zeta$ and a specification $\phi$ formal verification (e.g., [73]) proves or disproves that the system description $\zeta$ conforms to the specification $\phi$. If the system description $\zeta$ conforms to the specification $\phi$, then we say that $\phi$ holds in $\zeta$ or that $\phi$ is TRUE in $\zeta$. For a specification $\phi$ given as an LTL formula $\phi$ is TRUE in $\zeta$ iff every execution of $\zeta$ satisfies $\phi$. Note that if the system description itself is given as an LTL formula, then conformance of $\zeta$ to $\phi$ corresponds to implication: $\phi$ is TRUE in $\zeta$ iff $\zeta \rightarrow \phi$.

*Remark 13 (LTL Formal Verification as LTL Satisfiability)* Let $\zeta$ be a system description in LTL. Let $\phi$ be a specification in LTL. Then $\phi$ is TRUE in $\zeta$ iff $\zeta \wedge \neg\phi$ is unsatisfiable.

The fact that a system description conforms to a specification does not mean that all is well. An example is antecedent failure [8], where some specification $\phi \equiv \mathbf{G}(\psi \rightarrow \psi')$ holds in a system description $\zeta$, but the antecedent $\psi$ never becomes TRUE in any execution of $\zeta$. In that case the consequent $\psi'$ plays no role in determining that $\phi$ holds in $\zeta$. That often indicates presence of a problem in the specification or in the system description [9].

Vacuity generalizes that idea as follows [9, 65]. Let $\zeta$ be a system description, let $\phi$ be an LTL specification such that $\phi$ is TRUE in $\zeta$, and let $\psi$ be a positive (resp. negative) polarity occurrence of a subformula in $\phi$. Let $\phi'$ be obtained from $\phi$ by replacing $\psi$ with FALSE (resp. TRUE). If the modified specification $\phi'$ still holds in $\zeta$, then apparently $\psi$ has no influence on $\phi$ being TRUE in $\zeta$ in the following sense: $\psi$ could be replaced with any LTL formula in $\phi$ and the modified specification would still be TRUE in $\zeta$. In that case we say that $\phi$ is vacuously TRUE in $\zeta$. As an example consider a system $\zeta$ such that $p \wedge (\neg q) \wedge r \wedge (\neg s)$ is an invariant of $\zeta$, i.e., it holds in every reachable state of $\zeta$. Let the specification be $\phi \equiv \mathbf{G}((p \vee q) \wedge (r \vee s))$. $\phi$ is vacuously TRUE in $\zeta$, as $\phi$ is TRUE in $\zeta$ and either $q$ or $s$ could be replaced with FALSE without falsifying $\phi$ in $\zeta$.

Mutual vacuity [44] extends that idea to simultaneously replacing several subformulas with FALSE or TRUE depending on their polarity. In our example $q$ and $s$ could simultaneously be replaced with FALSE without falsifying $\phi$ in $\zeta$. This is not the case for any other pair of propositions in $\phi$. Definition 18 formalizes that notion. Proposition 5 then shows that the problems of determining mutual vacuity and of finding UCs in LTL can be reduced to each other.

**Definition 18 (Mutual Vacuity)** Let $\zeta$ be a system description. Let $\phi$ be a specification in LTL such that $\phi$ is TRUE in $\zeta$. Let $O$ be a set of disjoint subformula occurrences in $\phi$. Then $\phi$ is *mutually vacuously* TRUE in $O$ in $\zeta$ iff the modification $\phi'$ of $\phi$ that replaces those members of $O$ that have positive (resp. negative) polarity in $\phi$ with FALSE (resp. TRUE) is TRUE in $\zeta$.

**Proposition 5 (Reducibility between Mutual Vacuity and UCs in LTL)** *The problems of finding a UC of an unsatisfiable formula $\phi$ in LTL and of finding a set of subformula occurrences $O$ of an LTL specification $\phi$ that is* TRUE *in an LTL system description $\zeta$ such that $\phi$ is mutually vacuously* TRUE *in $O$ in $\zeta$ can be reduced to each other.*

*Proof* The proof is essentially by the respective definitions.

Assume that $\phi$ is mutually vacuously TRUE in $O$ in $\zeta$. Let $\phi'$ be $\phi$ with positive (resp. negative) polarity members of $O$ replaced with FALSE (resp. TRUE). Then (i) $\mu \equiv \zeta \wedge \neg\phi$ is unsatisfiable. (ii) $\mu' \equiv \zeta \wedge \neg\phi'$ is unsatisfiable. $\mu'$ is a UC of $\mu$ in LTL.

Assume that $\phi$ is an unsatisfiable LTL formula with UC $\phi^{uc}$. Then there exists a non-empty set of subformula occurrences $O'$ in $\phi$ such that $\phi^{uc}$ is obtained from $\phi$ by replacing positive (resp. negative) polarity members of $O'$ with TRUE (resp. FALSE). Now obviously (i) $\text{TRUE} \wedge \neg\neg\phi$ is unsatisfiable and (ii) $\text{TRUE} \wedge \neg\neg\phi^{uc}$ is unsatisfiable. I.e., $\neg\phi$ is mutually vacuously TRUE in $O'$ in the unconstrained system TRUE. □

A limited number of tools have been made available that can determine vacuity. `Aardvark` [77] computes — depending on configuration — maximal or maximum mutual vacuity for LTL using `VIS` [94] as a backend. `Aardvark` uses binary search and counterexamples to reduce the search space in the lattice of candidate strengthened specifications; it does not use proofs for passing specifications to obtain an initial candidate strengthened specification. Hence, our method is complementary. We performed a small set of trials with `Aardvark` on some of our benchmarks, mostly the smaller ones from each family. Within this, admittedly limited, set of trials we ran into problems with usability (often getting assertion violations or segmentation faults rather than error messages pointing to potential problems in our input) as well as scalability.[13] We therefore opted not to perform a comparison on the full set of benchmarks. `VaqTree` [91] implements the method of [92] that computes $k$-step vacuity for LTL, i.e., whether an occurrence of an atomic proposition is vacuous when bounded model checking runs only up to some bound $k$ are considered; the problem of removing the bound $k$ is left open. The `NuSMV Model Advisor`[14] computes the set of all occurrences of atomic propositions that are vacuous (but not necessarily mutually vacuous) for LTL. `VaqUoT` [41], a simplified implementation of [44], computes the set of all occurrences of atomic propositions that are vacuous (but not necessarily mutually vacuous) for CTL. A proof-based formulation of vacuity is suggested by Namjoshi [71]; no implementation or experiments are reported.

## 7 Experimental Evaluation

Our implementation, examples, and log files are available from `http://www.schuppan.de/viktor/actainformatica15/`.

### 7.1 Implementation

We implemented extraction of UCs as described in Sec. 4 in `TRP++`. We also implemented deletion-based minimization of UCs (Sec. 5.1) and grouping of propositions in UCs

[13] Note that in vacuity checking it is typically assumed that the system description is more complex than the specification, while in UC extraction all complexity is in the formula at hand. When (as we did in our trials) using the reduction from LTL UC extraction to vacuity checking from Prop. 5 the resulting vacuity checking instance consists of a trivial system description and a complex specification. In practice, when the vacuity checking procedure is tuned to take advantage of the small specification/complex system description scenario, then complex specifications may lead to problems; it seems that the scalability problems we observed with `Aardvark` are caused to some extent by the translation from the LTL specification into an explicit Büchi automaton performed in `VIS`.

[14] `http://code.google.com/a/eclipselabs.org/p/nusmv-tools/downloads/detail?name=NuSMVModelAdvisor_20121012.zip`

(Sec. 5.2). TRP++ provides a translation from LTL into SNF via an external tool. To facilitate tracing a UC in SNF back to the input formula in LTL we implemented a translator from LTL into SNF inside TRP++. We used parts of TSPASS[15] [67] for our implementation. For data structures we used C++ Standard Library containers (e.g., [57]), for graph operations the Boost Graph Library[16] [90].

Our translator from LTL into SNF reimplements ideas from the external translator, among them (i) normalizing and simplifying the LTL formula before translation, (ii) sharing the translation of several occurrences of the same subformula, and (iii) avoiding translation for SNF clauses in the input formula. In addition, we implemented pure literal simplification for LTL [22] for SNF clauses.

On the one hand, these optimizations are crucial for good performance. On the other hand, most optimizations change the input that is provided by the user before it is passed to our method for UC extraction, which potentially makes it harder for the user to understand how the UC she obtains corresponds to the input she provided. Therefore, for each such optimization we provide a command line option that disables the optimization. Mapping the optimized formula back to the input formula is left as future work.

Clearly, not solving a given LTL formula due to reaching run time or memory limits is the least useful result for a user. Hence, we assume the following usage model. A user will first enable all optimizations for the translation from LTL into SNF in order to solve a given LTL formula. If she cannot understand how the UC she obtains corresponds to the input she provided, she will selectively disable some of these optimizations. Notice that at that stage the user may already be able to exclude some parts of the LTL formula from consideration and, therefore, provide a reduced input to the solver. Based on these assumptions our implementation enables all optimizations for the translation from LTL into SNF by default. With one exception discussed next we used these default settings for all experiments reported in this article.

In previous versions of this work (including [85]) we used sharing of same polarity occurrences of a subformula in the translation from LTL into SNF. This turned out to negatively impact grouping of propositions in a UC. Hence, for better comparison all results reported in this article were obtained with sharing of same polarity occurrences of a subformula in the translation from LTL into SNF disabled. The data available from http://www.schuppan.de/viktor/actainformatica15/ include results with both sharing disabled and sharing enabled.

By default our implementation uses optimized resolution graphs to extract a UC (Def. 2, 3, 7, 8) and employs the optimizations in Rem. 6, 7. The results reported in this section were obtained with these optimizations enabled except in Sec. 7.5, where we evaluate the benefit of the optimizations.

## 7.2 Benchmarks

Our examples are based on [87]. In categories **crafted** and **random** and in family **forobots** we considered all unsatisfiable instances from [87]. The version of **alaska_lift** used here contains a small bug fix: in [29, 87] the subformula $\mathbf{X}u$ was erroneously written as literal $Xu$. Combining two variants of **alaska_lift** with three different scenarios we obtain six subfamilies of **alaska_lift**. For **anzu_genbuf** we invented three scenarios to obtain three subfamilies.

[15] http://www.csc.liv.ac.uk/~michel/software/tspass/
[16] http://www.boost.org/doc/libs/release/libs/graph/

**Table 3** Overview of benchmark families

| family | source | solved instances no UC extraction | (size of largest solved instance) | solved instances UC extraction | solved instances minimal UC extraction | solved instances grpd. prop. UC extraction |
|---|---|---|---|---|---|---|
| **application** | | | | | | |
| **alaska_lift** | [48, 29] | 70 | (4605) | 69 | 69 | 69 |
| **anzu_genbuf** | [15] | 15 | (1924) | 15 | 15 | 15 |
| **forobots** | [10] | 25 | (635) | 25 | 25 | 25 |
| **crafted** | | | | | | |
| **schuppan_O1formula** | [87] | 27 | (4006) | 27 | 27 | 27 |
| **schuppan_O2formula** | [87] | 8 | (91) | 8 | 8 | 8 |
| **schuppan_phltl** | [87] | 4 | (125) | 4 | 4 | 4 |
| **random** | | | | | | |
| **rozier_random** | [80] | 61 | (155) | 61 | 61 | 61 |
| **trp** | [55] | 397 | (1421) | 397 | 330 | 397 |

For benchmark families in categories **application** and **crafted**, which consist of sequences of instances of increasing difficulty, we stopped after two instances that could not be solved due to time or memory out. For benchmark families in category **random**, which consist of sequences of sets of instances of increasing size, we stopped after no instance was solved in three consecutive size levels. Some instances were simplified to FALSE during the translation from LTL into SNF; these instances were discarded.

In Tab. 3 we give an overview of the benchmark families. Columns 1 and 2 give the name and the source of the family. Columns 3, 5–7 list the numbers of instances that were solved by our implementation without UC extraction, with UC extraction, with minimal UC extraction, and with grouped propositions UC extraction. Column 4 indicates the size (as the number of nodes in the syntax tree) of the largest instance solved without UC extraction.

## 7.3 Setup

The experiments were performed on a laptop with an Intel Core i7 M 620 processor at 2 GHz running Ubuntu 14.04. Run time and memory usage were measured with `run`[17]. The time and memory limits were 600 seconds and 6 GB.

## 7.4 Extraction of UCs

In Fig. 6 (a), (b) we show the overhead that is incurred by extracting UCs as described in Sec. 4 over not extracting UCs. In Fig. 6 (c) we compare the sizes of the input formulas with the sizes of their UCs. For plots by category see App. B of [86].

Out of the 749 instances of all categories we considered with UC extraction disabled, 48 were simplified to FALSE in the translation into SNF, 607 were shown to be unsatisfiable by TR, and 94 remained unsolved. Enabling UC extraction results in one time out out of 607 instances.

The run time overhead for instances that take at least 0.7 seconds to solve without UC extraction, except for those of the **trp_N12y** subfamily, is at most 65 %. Instances of the

[17] http://fmv.jku.at/run/

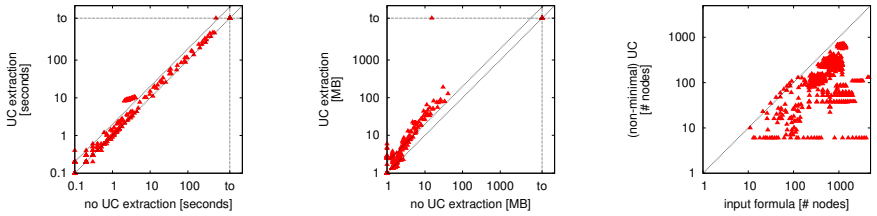



**Fig. 6** Comparison of UC extraction (y-axis) with no UC extraction (x-axis). (a) and (b) show the overhead incurred in terms of run time (in seconds) and memory (in MB). (c) shows the size reduction obtained, where size is measured as the number of nodes in the syntax trees. The off-center diagonal in (a) and (b) indicates where $y = 2x$.

**trp_N12y** subfamily incur a run time overhead between 150 and 300 %, which is the maximum overhead on our examples.

The memory overhead for instances that take at least 0.2 seconds to solve without UC extraction is at most 526 %. Note that peak memory usage with UC extraction was less than 200 MB.

In category **application** three subfamilies of family **alaska_lift** and all subfamilies of family **anzu_genbuf** have UCs of constant size, which are found by TRP++. The UCs of instances of these two families are at least 76 % smaller than the input formulas. Instances of family **forobots** are reduced between 47 % and 97 % with the median at 77 %. In category **crafted** family **schuppan_O1formula** also has UCs of constant size, which are found by TRP++. Instances of family **schuppan_O2formula** are themselves minimal UCs. Instances of **schuppan_phltl** have minimal UCs in which one top-level conjunct is removed from the input formula, which are found by TRP++ without minimization. In category **random** instances of family **rozier_random** are reduced between 6 % and 95 % with the median at 75 %. Instances of family **trp** exhibit minimum, median, and maximum reductions of 26 %, 57 %, and 88 %, respectively.

Our data show that extraction of UCs is possible with quite acceptable overhead in run time and memory usage. The resulting UCs are often significantly smaller than the input formula.

### 7.5 Optimizations

In Fig. 7 we show the benefit of the optimizations described in Rem. 6 and in Sec. 4.1.2 when extracting UCs. We show the impact on the peak size of the resolution graph rather than on run time or memory, as the former is implementation independent.

The impact of including premise 1 of BFS-loop-it-init-c during the construction of the resolution graph and disabling immediate pruning of vertices and edges in partitions of failed loop search iterations from the resolution graph in Fig. 7 (b) (the former implies the latter) and of disabling pruning non-reachable vertices from the resolution graph between loop searches in Fig. 7 (e) is quite significant. In Fig. 7 (b) the median increase of the peak size of the resolution graph is 39 %, the maximum is 1339 %. In Fig. 7 (e) slightly more than half of all instances exhibit no change in the size of the resolution graph; for most of the remaining instances the peak size of the resolution graph increases by 20 % or more with the maximum at 858 %.

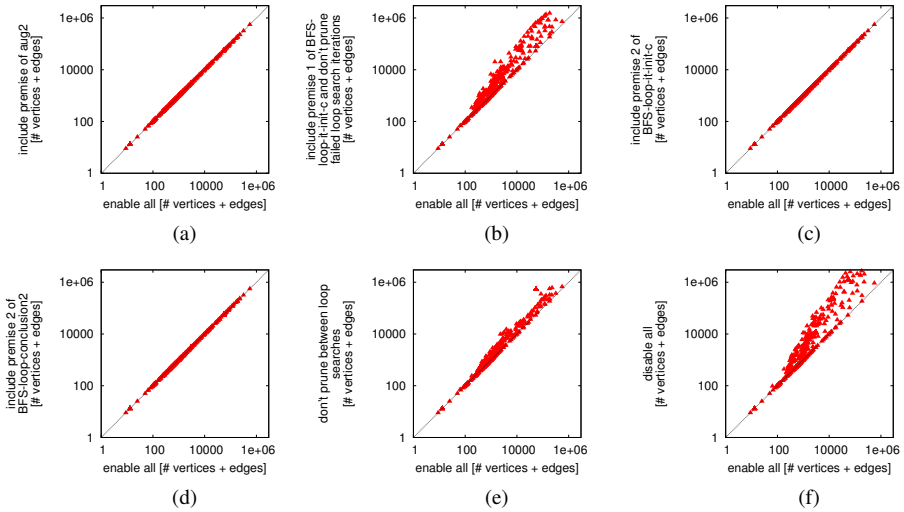



**Fig. 7** Benefit of optimizations as reduction in the peak size of the resolution graph (number of vertices + number of edges). The x-axis shows all optimizations enabled. The y-axis of (a)–(e) shows one optimization disabled: (a) include premise of aug2, (b) include premise 1 of BFS-loop-it-init-c and disable immediate pruning of failed loop search iterations, (c) include premise 2 of BFS-loop-it-init-c, (d) include premise 2 of BFS-loop-conclusion2, (e) disable pruning of the resolution graph between loop searches. The y-axis of (f) shows all optimizations disabled.

The impact in the remaining cases (Fig. 7 (a), (c), (d)) is negligible: for no instance the peak size of the resolution graph increases by more than 6 %. However, in cases (c) and (d) there is an instance where disabling the optimization leads to a larger UC. This occurs more often also in case (b).

We note that in each family of categories **application** and **random** there are instances for which the memory overhead of disabling all optimizations is larger than 1000 %. The highest memory overhead is more than 6500 % for an instance of family **alaska_lift**. The effect on run time is less pronounced and uniform. When only instances are taken into account that take at least 0.3 seconds to solve with all optimizations enabled, then the run time overhead of disabling the optimizations is at most 40 %.

## 7.6 Extraction of Minimal UCs

Figure 8 shows the costs and benefits of applying deletion-based minimization (Sec. 5.1) to (non-minimal) UCs obtained as described in Sec. 4.

Costs and benefits are somewhat varied. Minimal UCs can be computed for all instances for which (non-minimal) UCs were obtained except for all 67 instances in family **trp_N12y**.

A closer analysis shows that most instances with reductions of 30 % or more are in the **random** category; some are also in family **forobots**. The largest reduction seen is 62 % from 469 to 177 nodes in the syntax tree of an instance from family **trp_N12x**. 5 instances of the **forobots** family are reduced between 30 % and 57 %. In the **application** category several instances in each of the three families exhibit reductions in the range between 20 % and 28 %. On the other hand, for three out of six subfamilies in the **alaska_lift** family, for one

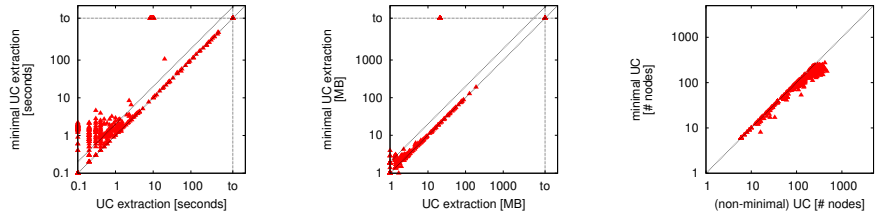



**Fig. 8** Comparison of minimal UC extraction (y-axis) with UC extraction (x-axis). (a) and (b) show the overhead incurred in terms of run time (in seconds) and memory (in MB). (c) shows the size reduction obtained, where size is measured as the number of nodes in the syntax trees. The off-center diagonal in (a) and (b) indicates where $y = 2x$.

out of three subfamilies in the **anzu_genbuf** family, and for all families in the the **crafted** category a minimal UC was already obtained without deletion-based minimization (for the **schuppan_O2formula** family the original instances are minimal UCs.).

The run time (resp. memory) overhead to obtain a minimal UC, except for instances of the **trp** family that take 0.5 seconds or less to solve without deletion-based minimization, is at most 428 % (resp. 87 %).

Hence, while based on our data there is no simple conclusion regarding the costs and benefits of applying deletion-based minimization, one should keep in mind that minimizing UCs trades increased processing time by a computer for reduced analysis time by a human user, and that a minimal UC has less potential to confuse a user about how parts of a UC contribute to its unsatisfiability.

### 7.7 UCs with Grouped Propositions

In Tab. 4 we show the effect of grouping propositions in UCs. The first column gives the name of the family. The second column shows the maximum number of groups that the set of occurrences of some proposition in the UC of a member of this benchmark family was partitioned into. The number in parentheses shows the number of propositions (not occurrences of propositions!) in that UC. The third column indicates the maximum number of propositions whose occurrences in the UC were partitioned into two or more groups for a member of this benchmark family. The number in parentheses is as for the previous column.

Our data show that instances from the **application** and **random** categories exhibit significant potential for grouping propositions in UCs. The **schuppan_O1formula** and **schuppan_phltl** families in the **crafted** category provide no opportunity for grouping propositions. Manual inspection of the groupings obtained in the **crafted** category proves them to be optimal.

(25) shows the UC with grouped propositions obtained from instance **lift_b_l3_3** in family **alaska_lift**.[18] We use superscripts $^0, ^1, ^2$ to distinguish groups of occurrences of the same proposition. The example illustrates clearly that grouped propositions make it easier to see

[18] The duplicate occurrence of $\neg bf_1^1$ in line 2 of (25) is an artifact resulting from the simplification of UCs by removing conjunction with TRUE and disjunction with FALSE.

**Table 4** Effect of grouping propositions in UCs

| family | max. # groups per prop. in UC (# prop. in UC) | max. # prop. w. $\geq 2$ groups per prop. in UC (# prop. in UC) |
|---|---|---|
| **application** | | |
| **alaska_lift** | 3 (9) | 6 (18) |
| **anzu_genbuf** | 2 (4) | 1 (4) |
| **forobots** | 2 (4) | 4 (18) |
| **crafted** | | |
| **schuppan_O1formula** | 1 (1) | 0 (1) |
| **schuppan_O2formula** | 2 (3) | 8 (17) |
| **schuppan_phltl** | 1 (2) | 0 (2) |
| **random** | | |
| **rozier_random** | 4 (4) | 2 (6) |
| **trp** | 7 (27) | 10 (27) |

which occurrences of propositions interact.

$$
\begin{aligned}
&(\neg u^0) \wedge (\neg up) \wedge (\neg bf_0^0) \wedge (\neg bf_1^0) \wedge \\
&(\mathbf{G}(bf_0^0 \vee bf_1^0 \vee \mathbf{X}((\neg bf_1^1) \vee \neg bf_1^1))) \wedge \\
&(\mathbf{G}(bf_0^1 \vee (\neg u^1) \vee \neg \mathbf{X} bf_0^2)) \wedge \\
&(\mathbf{G}(bf_1^1 \vee (\neg u^1) \vee \neg \mathbf{X} bf_1^2)) \wedge \\
&(\mathbf{G}(up \vee bf_0^0 \vee bf_1^0 \vee \neg \mathbf{X}(bf_0^1 \wedge \neg bf_1^1))) \wedge \\
&(\mathbf{G}((\neg \mathbf{X} u^1) \rightarrow u^0)) \wedge \\
&\mathbf{XX}(bf_0^2 \vee bf_1^2)
\end{aligned}
\tag{25}
$$

In Fig. 9 we show the overhead that is incurred by the extraction of UCs with grouped propositions over the extraction of UCs without grouped propositions.

The run time (resp. memory) overhead for instances that take at least 0.4 seconds to solve without grouped propositions is at most 24 % (resp. 111 %). The memory overhead is less than 30 % for more than 93 % of all instances.

The moderate run time and memory overhead seems justified given the potential benefits of grouped propositions illustrated by the example above.

## 7.8 Comparison with Related Approaches

We now discuss `TRP++` in comparison to `PLTL-MUP` [43] and `procmine` [3]. Remember that both `PLTL-MUP` and `procmine` compute an unsatisfiable subset of a set of LTL formulas

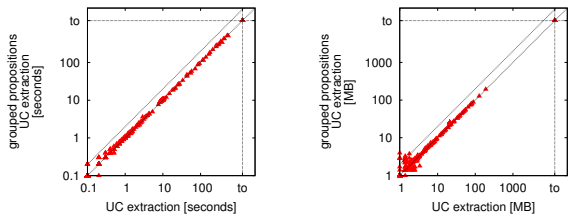



**Fig. 9** Overhead of UC extraction with grouped propositions (y-axis) over UC extraction (x-axis) in terms of run time (in seconds, (a)) and memory (in MB, (b)). The off-center diagonal indicates where $y = 2x$.

rather than a UC in LTL as a syntax tree according to Def. 11. I.e., compared to our notion of UC PLTL-MUP and procmine likely explore a smaller search space during minimization (if enabled) and possibly produce a less fine-grained result.

TRP++, PLTL-MUP, and procmine differ in important respects such as preprocessing of the input formula, algorithm used to obtain a UC, algorithm used to minimize a UC, programming language used for the implementation, and — last but not least — the notion of UC (syntax tree vs. subset of a set of formulas). Therefore, we regard the value of a detailed discussion of the performance of TRP++ compared to PLTL-MUP and procmine as somewhat questionable. We only state that, when using the options for TRP++ as explained above and default options for PLTL-MUP and procmine, and minimization in TRP++ enabled iff it was enabled in its competitor (i) TRP++ mostly outperforms PLTL-MUP and procmine in category **application** and in family **trp** of category **random**, (ii) in category **crafted** TRP++ is outperformed by PLTL-MUP in one out of three families and by procmine in most instances, and (iii) in family **rozier_random** of category **random** neither tool clearly dominates an other. For plots please refer to App. B of [86], for data to http://www.schuppan.de/viktor/actainformatica15/.

We finally illustrate the difference between our notion of UC and the notion of UC used by PLTL-MUP and procmine on an example. (26) shows the UC produced by procmine for instance **liftf_l1_2** from family **alaska_lift**. The UC contains 6 out of 15 top level conjuncts. The parts that are additionally replaced with TRUE by TRP++ due to our more fine-grained notion of UC are marked blue boxed. For PLTL-MUP or procmine to obtain a similar result, the user would have to rewrite the input. More such examples can be found in families **alaska_lift**, **forobots**, and **rozier_random** (PLTL-MUP and procmine solve no instance in the remaining family in category **application**, **anzu_genbuf**).

$$
\begin{array}{l}
f_0, \\
\neg up, \\
\mathbf{G}(\boxed{(u \rightarrow ((f_0 \rightarrow \mathbf{X}f_0) \wedge ((\mathbf{X}f_0) \rightarrow f_0) \wedge (f_1 \rightarrow \mathbf{X}f_1) \wedge ((\mathbf{X}f_1) \rightarrow f_1)))} \wedge \\
\quad (f_0 \rightarrow \mathbf{X}(f_0 \vee f_1)) \wedge \\
\quad \boxed{(f_1 \rightarrow \mathbf{X}(f_0 \vee f_1))}), \\
\mathbf{G}(((f_0 \wedge \mathbf{X}f_0) \rightarrow (\boxed{(up \rightarrow \mathbf{X}up)} \wedge ((\mathbf{X}up) \rightarrow up))) \wedge \\
\quad \boxed{((f_1 \wedge \mathbf{X}f_1) \rightarrow ((up \rightarrow \mathbf{X}up) \wedge ((\mathbf{X}up) \rightarrow up)))} \wedge \\
\quad ((f_0 \wedge \mathbf{X}f_1) \rightarrow up) \wedge \\
\quad \boxed{((f_1 \wedge \mathbf{X}f_0) \rightarrow \neg up)}), \\
\mathbf{G}(\boxed{(b_0 \rightarrow \mathbf{F}f_0)} \wedge (b_1 \rightarrow \mathbf{F}f_1)), \\
\mathbf{GF}b_1
\end{array}
\tag{26}
$$

All in all TRP++ produces more fine-grained UCs than PLTL-MUP and procmine while remaining at least competitive in terms of run time and memory usage.

## 8 Conclusions

In this article we showed how to obtain UCs for LTL via temporal resolution, and we demonstrated with an implementation in TRP++ that UC extraction can be performed efficiently. The resulting UCs are significantly smaller than the corresponding input formulas and more fine-grained than those produced by existing tools. In parallel work [84] this article has been used as a basis to suggest enhancing UCs for LTL with information on when subformulas of a UC are relevant for unsatisfiability. The similarity of temporal resolution and some

BDD-based algorithms at a high level and work on resolution with BDDs ([59]) suggests to explore whether computation of UCs is feasible for BDD-based algorithms. Another direction for transfer of our results is resolution-based computation of unrealizable cores [72].

**Acknowledgements** I am grateful to Boris Konev and Michel Ludwig for making `TRP++` and `TSPASS` (which are the basis of this paper) including their LTL translators available and for answering my questions. I thank Rajeev Goré, Zhe Hou, Timothy Sergeant, and Jimmy Thomson for availability of and discussion about `PLTL-MUP` and `procmine` as well as Daniel Kroening, Mitra Purandare, and Thomas Wahl for availability of and discussion about `Aardvark`. I thank Alessandro Cimatti for bringing up the subject of temporal resolution. I also thank the reviewers of the current and previous iterations of this article for their helpful feedback. Initial parts of the work were performed while working under a grant by the Provincia Autonoma di Trento (project EMTELOS).

## References

1. Amjad H (2006) Compressing Propositional Refutations. In: Merz S, Nipkow T (eds) AVoCS, Elsevier, Electr. Notes Theor. Comput. Sci., vol 185, pp 3–15
2. Armoni R, Fix L, Flaisher A, Grumberg O, Piterman N, Tiemeyer A, Vardi M (2003) Enhanced Vacuity Detection in Linear Temporal Logic. In: Hunt Jr W, Somenzi F (eds) CAV, Springer, LNCS, vol 2725, pp 368–380
3. Awad A, Goré R, Hou Z, Thomson J, Weidlich M (2012) An iterative approach to synthesize business process templates from compliance rules. Inf Syst 37(8):714–736
4. Baader F, Calvanese D, McGuinness D, Nardi D, Patel-Schneider P (eds) (2007) The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press
5. Bachmair L, Ganzinger H (2001) Resolution Theorem Proving. In: Robinson J, Voronkov A (eds) Handbook of Automated Reasoning, Elsevier and MIT Press, pp 19–99
6. Bakker R, Dikker F, Tempelman F, Wognum P (1993) Diagnosing and Solving Over-Determined Constraint Satisfaction Problems. In: IJCAI, pp 276–281
7. Barrett C, Sebastiani R, Seshia S, Tinelli C (2009) Satisfiability Modulo Theories. In: [14], pp 825–885
8. Beatty D, Bryant R (1994) Formally Verifying a Microprocessor Using a Simulation Methodology. In: DAC, pp 596–602
9. Beer I, Ben-David S, Eisner C, Rodeh Y (2001) Efficient Detection of Vacuity in Temporal Model Checking. Formal Methods in System Design 18(2):141–163
10. Behdenna A, Dixon C, Fisher M (2009) Deductive Verification of Simple Foraging Robotic Behaviours. International Journal of Intelligent Computing and Cybernetics 2(4):604–643
11. Belov A, Marques Silva J (2011) Minimally Unsatisfiable Boolean Circuits. In: Sakallah K, Simon L (eds) SAT, Springer, LNCS, vol 6695, pp 145–158
12. Biere A (2009) Bounded Model Checking. In: [14], pp 457–481
13. Biere A, Heljanko K, Junttila T, Latvala T, Schuppan V (2006) Linear Encodings of Bounded LTL Model Checking. Logical Methods in Computer Science 2(5)
14. Biere A, Heule M, van Maaren H, Walsh T (eds) (2009) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol 185, IOS Press
15. Bloem R, Galler S, Jobstmann B, Piterman N, Pnueli A, Weiglhofer M (2007) Specify, Compile, Run: Hardware from PSL. In: Glesner S, Knoop J, Drechsler R (eds) COCV, Elsevier, Electr. Notes Theor. Comput. Sci., vol 190(4), pp 3–16

16. Bruni R, Sassano A (2001) Restoring Satisfiability or Maintaining Unsatisfiability by finding small Unsatisfiable Subformulae. In: Kautz H, Selman B (eds) SAT, Elsevier, Electronic Notes in Discrete Mathematics, vol 9, pp 162–173
17. Büning HK, Kullmann O (2009) Minimal Unsatisfiability and Autarkies. In: [14], pp 339–401
18. Burch J, Clarke E, McMillan K, Dill D, Hwang L (1992) Symbolic Model Checking: $10^{20}$ States and Beyond. Inf Comput 98(2):142–170
19. Chiappini A, Cimatti A, Macchi L, Rebollo O, Roveri M, Susi A, Tonetta S, Vittorini B (2010) Formalization and validation of a subset of the European Train Control System. In: Kramer J, Bishop J, Devanbu P, Uchitel S (eds) ICSE (2), ACM, pp 109–118
20. Chinneck J, Dravnieks E (1991) Locating Minimal Infeasible Constraint Sets in Linear Programs. INFORMS Journal on Computing 3(2):157–168
21. Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma E, Larsen K (eds) CAV, Springer, LNCS, vol 2404, pp 359–364
22. Cimatti A, Roveri M, Schuppan V, Tonetta S (2007) Boolean Abstraction for Temporal Logic Satisfiability. In: Damm W, Hermanns H (eds) CAV, Springer, LNCS, vol 4590, pp 532–546
23. Cimatti A, Roveri M, Schuppan V, Tchaltsev A (2008) Diagnostic Information for Realizability. In: Logozzo F, Peled D, Zuck L (eds) VMCAI, Springer, LNCS, vol 4905, pp 52–67
24. Cimatti A, Griggio A, Sebastiani R (2011) Computing Small Unsatisfiable Cores in Satisfiability Modulo Theories. J Artif Intell Res (JAIR) 40:701–728
25. Cimatti A, Mover S, Tonetta S (2011) Proving and explaining the unfeasibility of message sequence charts for hybrid systems. In: Bjesse P, Slobodová A (eds) FMCAD, FMCAD Inc., pp 54–62
26. Clarke E, Grumberg O, Hamaguchi K (1997) Another Look at LTL Model Checking. Formal Methods in System Design 10(1):47–71
27. Clarke E, Grumberg O, Peled D (2001) Model checking. MIT Press
28. Clarke E, Talupur M, Veith H, Wang D (2003) SAT Based Predicate Abstraction for Hardware Verification. In: Giunchiglia E, Tacchella A (eds) SAT, Springer, LNCS, vol 2919, pp 78–92
29. De Wulf M, Doyen L, Maquet N, Raskin J (2008) Antichains: Alternative Algorithms for LTL Satisfiability and Model-Checking. In: Ramakrishnan C, Rehof J (eds) TACAS, Springer, LNCS, vol 4963, pp 63–77
30. Demri S, Schnoebelen P (2002) The Complexity of Propositional Linear Temporal Logics in Simple Cases. Inf Comput 174(1):84–103
31. Dixon C (1995) Strategies for temporal resolution. PhD thesis, Department of Computer Science, University of Manchester, available from `http://apt.cs.manchester.ac.uk/ftp/pub/TR/UMCS-95-12-1.ps.Z`
32. Dixon C (1997) Using Otter for Temporal Resolution. In: Barringer H, Fisher M, Gabbay D, Gough G (eds) ICTL, Springer, Applied Logic Series, vol 16, pp 149–166
33. Dixon C (1998) Temporal Resolution Using a Breadth-First Search Algorithm. Ann Math Artif Intell 22(1-2):87–115
34. D'Silva V, Kroening D, Purandare M, Weissenbacher G (2010) Interpolant Strength. In: Barthe G, Hermenegildo M (eds) VMCAI, Springer, LNCS, vol 5944, pp 129–145
35. Eisner C, Fisman D (2006) A Practical Introduction to PSL. Springer
36. Emerson E (1990) Temporal and Modal Logic. In: van Leeuwen J (ed) Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics, Elsevier and

MIT Press, pp 995–1072
37. Fisher M (1991) A Resolution Method for Temporal Logic. In: IJCAI, pp 99–104
38. Fisher M, Noël P (1992) Transformation and synthesis in METATEM. Part I: Propositional METATEM. Tech. Rep. UMCS-92-2-1, University of Manchester, Department of Computer Science, available from http://apt.cs.manchester.ac.uk/ftp/pub/TR/UMCS-92-2-1.ps.Z
39. Fisher M, Dixon C, Peim M (2001) Clausal temporal resolution. ACM Trans Comput Log 2(1):12–56
40. Fisman D, Kupferman O, Sheinvald-Faragy S, Vardi M (2008) A Framework for Inherent Vacuity. In: Chockler H, Hu A (eds) HVC, Springer, LNCS, vol 5394, pp 7–22
41. Gheorghiu M, Gurfinkel A (2006) VaqUoT: A tool for vacuity detection. In: Misra J, Nipkow T, Sekerinski E (eds) FM, Springer, LNCS, vol 4085, tool presentation. Available from http://fm06.mcmaster.ca/VaqUoT.pdf.
42. Goldberg E, Novikov Y (2003) Verification of Proofs of Unsatisfiability for CNF Formulas. In: DATE, IEEE Computer Society, pp 10,886–10,891
43. Goré R, Huang J, Sergeant T, Thomson J (2013) Finding Minimal Unsatisfiable Subsets in Linear Temporal Logic using BDDs. Available from http://www.timsergeant.com/files/pltlmup/gore_huang_sergeant_thomson_mus_pltl.pdf
44. Gurfinkel A, Chechik M (2004) How Vacuous Is Vacuous? In: Jensen K, Podelski A (eds) TACAS, Springer, LNCS, vol 2988, pp 451–466
45. Halpern J, Reif J (1983) The Propositional Dynamic Logic of Deterministic, Well-Structured Programs. Theor Comput Sci 27:127–165
46. Hantry F, Hacid M (2011) Handling Conflicts in Depth-First Search for LTL Tableau to Debug Compliance Based Languages. In: Pimentel E, Valero V (eds) FLACOS, EPTCS, vol 68, pp 39–53
47. Hantry F, Saïs L, Hacid M (2012) On the complexity of computing minimal unsatisfiable LTL formulas. Electronic Colloquium on Computational Complexity (ECCC) 19(69)
48. Harding A (2005) Symbolic Strategy Synthesis For Games With LTL Winning Conditions. PhD thesis, University of Birmingham
49. Heuerding A, Jäger G, Schwendimann S, Seyfried M (1995) Propositional Logics on the Computer. In: Baumgartner P, Hähnle R, Posegga J (eds) TABLEAUX, Springer, LNCS, vol 918, pp 310–323
50. Hoos H (1999) Heavy-Tailed Behaviour in Randomised Systematic Search Algorithms for SAT? Tech. Rep. TR-99-16, University of British Columbia, Department of Computer Science
51. Horridge M (2011) Justification based explanation in ontologies. PhD thesis, School of Computer Science, Faculty of Engineering and Physical Sciences, University of Manchester, available from http://www.escholar.manchester.ac.uk/uk-ac-man-scw:131699
52. Huang J (2005) MUP: a minimal unsatisfiability prover. In: Tang T (ed) ASP-DAC, ACM Press, pp 432–437
53. Hustadt U, Konev B (2003) TRP++ 2.0: A Temporal Resolution Prover. In: Baader F (ed) CADE, Springer, LNCS, vol 2741, pp 274–278
54. Hustadt U, Konev B (2004) TRP++: A temporal resolution prover. In: Baaz M, Makowsky J, Voronkov A (eds) Collegium Logicum, vol 8, Kurt Gödel Society, pp 65–79

55. Hustadt U, Schmidt RA (2002) Scientific Benchmarking with Temporal Logic Decision Procedures. In: Fensel D, Giunchiglia F, McGuinness D, Williams M (eds) KR, Morgan Kaufmann, pp 533–546
56. Jobstmann B, Bloem R (2006) Optimizations for LTL Synthesis. In: FMCAD, IEEE Computer Society, pp 117–124
57. Josuttis N (2012) The C++ Standard Library: A Tutorial and Reference, 2nd edn. Addison-Wesley
58. Junker U (2001) QuickXplain: Conflict detection for arbitrary constraint propagation algorithms. In: CONS, available from `http://www.lirmm.fr/~bessiere/ws_ijcai01/junker.ps.gz`
59. Jussila T, Sinz C, Biere A (2006) Extended Resolution Proofs for Symbolic SAT Solving with Quantification. In: Biere A, Gomes C (eds) SAT, Springer, LNCS, vol 4121, pp 54–60
60. Kesten Y, Pnueli A, Raviv L (1998) Algorithmic Verification of Linear Temporal Logic Specifications. In: Larsen K, Skyum S, Winskel G (eds) ICALP, Springer, LNCS, vol 1443, pp 1–16
61. Könighofer R, Hofferek G, Bloem R (2009) Debugging formal specifications using simple counterstrategies. In: FMCAD, IEEE, pp 152–159
62. Könighofer R, Hofferek G, Bloem R (2010) Debugging Unrealizable Specifications with Model-Based Diagnosis. In: Barner S, Harris I, Kroening D, Raz O (eds) HVC, Springer, LNCS, vol 6504, pp 29–45
63. Kress-Gazit H, Fainekos G, Pappas G (2008) Translating Structured English to Robot Controllers. Advanced Robotics 22(12):1343–1359
64. Kupferman O (2006) Sanity Checks in Formal Verification. In: Baier C, Hermanns H (eds) CONCUR, Springer, LNCS, vol 4137, pp 37–51
65. Kupferman O, Vardi M (2003) Vacuity detection in temporal model checking. STTT 4(2):224–233
66. Lichtenstein O, Pnueli A (1985) Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In: Van Deusen M, Galil Z, Reid B (eds) POPL, ACM Press, pp 97–107
67. Ludwig M, Hustadt U (2010) Implementing a fair monodic temporal logic prover. AI Commun 23(2-3):69–96
68. Marques-Silva J (2012) Computing minimally unsatisfiable subformulas: State of the art and future directions. Multiple-Valued Logic and Soft Computing 19(1-3):163–183
69. Marques-Silva J, Janota M (2014) Computing minimal sets on propositional formulae i: Problems & reductions. Available from `arXiv:1402.3011 [cs.LO]`
70. Nadel A (2009) Understanding and improving a modern SAT solver. PhD thesis, The Blavatnik School of Computer Science, Raymond and Beverly Sackler Faculty of Exact Sciences, Tel Aviv University, available from `http://www.cs.tau.ac.il/thesis/thesis/nadel.pdf`
71. Namjoshi K (2004) An Efficiently Checkable, Proof-Based Formulation of Vacuity in Model Checking. In: Alur R, Peled D (eds) CAV, Springer, LNCS, vol 3114, pp 57–69
72. Noël P (1995) A Transformation-Based Synthesis of Temporal Specifications. Formal Asp Comput 7(6):587–619
73. Peled D (2001) Software Reliability Methods. Texts in Computer Science, Springer
74. Pesic M, van der Aalst W (2006) A Declarative Approach for Flexible Business Processes Management. In: Eder J, Dustdar S (eds) Business Process Management Workshops, Springer, LNCS, vol 4103, pp 169–180

75. Pill I, Semprini S, Cavada R, Roveri M, Bloem R, Cimatti A (2006) Formal analysis of hardware requirements. In: Sentovich E (ed) DAC, ACM, pp 821–826
76. Plaisted D, Greenbaum S (1986) A Structure-Preserving Clause Form Translation. J Symb Comput 2(3):293–304
77. Purandare M, Wahl T, Kroening D (2009) Strengthening properties using abstraction refinement. In: DATE, IEEE, pp 1692–1697
78. Raman V, Kress-Gazit H (2011) Analyzing Unsynthesizable Specifications for High-Level Robot Behavior Using LTLMoP. In: Gopalakrishnan G, Qadeer S (eds) CAV, Springer, LNCS, vol 6806, pp 663–668
79. Rollini S, Bruttomesso R, Sharygina N, Tsitovich A (2014) Resolution proof transformation for compression and interpolation. Formal Methods in System Design 45(1):1–41
80. Rozier K, Vardi M (2010) LTL satisfiability checking. STTT 12(2):123–137
81. Schlobach S, Cornet R (2003) Non-Standard Reasoning Services for the Debugging of Description Logic Terminologies. In: Gottlob G, Walsh T (eds) IJCAI, Morgan Kaufmann, pp 355–362
82. Schuppan V (2012) Extracting unsatisfiable cores for LTL via temporal resolution. Available at `arXiv:1212.3884v1 [cs.LO]`
83. Schuppan V (2012) Towards a notion of unsatisfiable and unrealizable cores for LTL. Sci Comput Program 77(7-8):908–939
84. Schuppan V (2013) Enhancing Unsatisfiable Cores for LTL with Information on Temporal Relevance. In: Bortolussi L, Wiklicky H (eds) QAPL, EPTCS, vol 117, pp 49–65
85. Schuppan V (2013) Extracting Unsatisfiable Cores for LTL via Temporal Resolution. In: Sanchez C, Venable B, Zimanyi E (eds) TIME, IEEE Computer Society, pp 54–61
86. Schuppan V (2015) Extracting unsatisfiable cores for LTL via temporal resolution (full version). Available at `arXiv:1212.3884 [cs.LO]`
87. Schuppan V, Darmawan L (2011) Evaluating LTL Satisfiability Solvers. In: Bultan T, Hsiung P (eds) ATVA, Springer, LNCS, vol 6996, pp 397–413
88. Shlyakhter I (2005) Declarative symbolic pure-logic model checking. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusets Institute of Technology, available from `http://hdl.handle.net/1721.1/30184`
89. Shlyakhter I, Seater R, Jackson D, Sridharan M, Taghdiri M (2003) Debugging Over-constrained Declarative Models Using Unsatisfiable Cores. In: ASE, IEEE Computer Society, pp 94–105
90. Siek J, Lee L, Lumsdaine A (2002) The Boost Graph Library - User Guide and Reference Manual. C++ in-depth series, Pearson / Prentice Hall
91. Simmonds J, Davies J, Gurfinkel A (2006) VaqTree: Efficient vacuity detection for bounded model checking. In: Misra J, Nipkow T, Sekerinski E (eds) FM, Springer, LNCS, vol 4085, tool presentation. Available from `http://fm06.mcmaster.ca/jocelyn.pdf`.
92. Simmonds J, Davies J, Gurfinkel A, Chechik M (2010) Exploiting resolution proofs to speed up LTL vacuity detection for BMC. STTT 12(5):319–335
93. Sistla A, Clarke E (1985) The Complexity of Propositional Linear Temporal Logics. J ACM 32(3):733–749
94. The VIS Group (1996) VIS: A System for Verification and Synthesis. In: Alur R, Henzinger T (eds) CAV, Springer, LNCS, vol 1102, pp 428–432
95. Torlak E, Chang FSH, Jackson D (2008) Finding Minimal Unsatisfiable Cores of Declarative Specifications. In: Cuéllar J, Maibaum T, Sere K (eds) FM, Springer, LNCS, vol 5014, pp 326–341

96. Van Gelder A (2002) Extracting (Easily) Checkable Proofs from a Satisfiability Solver that Employs both Preorder and Postorder Resolution, AI&M 24-2002. Seventh International Symposium on Artificial Intelligence and Mathematics, January 2-4, 2002, Fort Lauderdale, Florida
97. Whalley D (1994) Automatic Isolation of Compiler Errors. ACM Trans Program Lang Syst 16(5):1648–1659
98. Zeller A, Hildebrandt R (2002) Simplifying and Isolating Failure-Inducing Input. IEEE Trans Software Eng 28(2):183–200
99. Zhang L (2003) Searching for Truth: Techniques for Satisfiability of Boolean Formulas. PhD thesis, Department of Electrical Engineering, Princeton University
100. Zhang L, Malik S (2003) Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In: DATE, IEEE Computer Society, pp 10,880–10,885